CoCentric®
SystemC™ Compiler
RTL User and Modeling Guide

Version 2001.08, August 2001

Comments?
E-mail your comments about Synopsys documentation to doc@synopsys.com

SYNOPSYS®

Copyright Notice and Proprietary Information

Copyright © 2001 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Right to Copy Documentation

The license agreement with Synopsys permits licensee to make copies of the documentation for its internal use only. Each copy shall include all copyrights, trademarks, service marks, and proprietary rights notices, if any. Licensee must assign sequential numbers to all copies. These copies shall contain the following legend on the cover page:

> "This document is duplicated with the permission of Synopsys, Inc., for the exclusive use of _____ and its employees. This is copy number _____."

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, the Synopsys logo, AMPS, Arcadia, CoCentric, COSSAP, Cyclone, DelayMill, DesignPower, DesignSource, DesignWare, Eagle Design, EPIC, Formality, in-Sync, Learn-It!, Logic Automation, Logic Modeling, ModelAccess, ModelTools, PathMill, PowerArc, PowerMill, PrimeTime, RailMill, SmartLogic, SmartModel, SmartModels, SNUG, Solv-It, SolvNet, Stream Driven Simulator, TestBench Manager, TetraMAX, TimeMill, and VERA are registered trademarks of Synopsys, Inc.

Trademarks (™)

BCView, Behavioral Compiler, BOA, BRT, Cedar, DC Expert, DC Expert *Plus*, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Compiler, DesignTime, Direct RTL, Direct Silicon Access, dont_use, DW8051, DWPCI, ECL Compiler, ECO Compiler, ExpressModel, Floorplan Manager, FoundryModel, FPGA Compiler II, FPGA *Express*, Frame Compiler, HDL Advisor, HDL Compiler, Integrator, Interactive Waveform Viewer, LEDA, Liberty, Library Compiler, ModelSource, Module Compiler, MS-3200, MS-3400, NanoSim, Physical Compiler, Power Compiler, PowerCODE, PowerGate, ProFPGA, Protocol Compiler, RoadRunner, RTL Analyzer, Schematic Compiler, Scirocco, Shadow Debugger, SmartLicense, SmartModel Library, Source-Level Design, SWIFT, Synopsys Eagle Design Automation, Synopsys Eagle*i*, Synopsys EagleV, Synthetic Designs, SystemC, Test Compiler, TestGen, TimeTracker, Timing Annotator, Trace-On-Demand, VCS, VCS Express, VCSi, VHDL Compiler, VHDL System Simulator, VirSim, VMC, and VSS are trademarks of Synopsys, Inc.

Service Marks (SM)

DesignSphere and TAP-in are service marks of Synopsys, Inc.

All other product or company names may be trademarks of their respective owners.

Printed in the U.S.A.

Document Order Number: 37582-000 KA
SystemC Compiler RTL User and Modeling Guide, v2001.08

Contents

What's New in This Release xxii

About This Guide. xxv

Customer Support xxviii

1. Using SystemC Compiler for RTL Synthesis

Synthesis With SystemC Compiler 1-3

Choosing the Right Abstraction for Synthesis 1-4

Identifying Attributes Suitable for RTL Synthesis 1-4

Identifying Attributes Suitable for Behavioral Synthesis 1-5

Refinement Overview 1-6

Inputs and Outputs for RTL Synthesis. 1-7

RTL Description 1-8

Technology Library. 1-8

Synthetic Library 1-9

Outputs From SystemC Compiler 1-9

Synthesizing a Single RTL Module 1-10

Starting SystemC Compiler 1-11

Analyzing and Elaborating a SystemC RTL Module 1-11
Creating an Elaborated .db File for Synthesis 1-12
Creating an RTL HDL Description 1-13
Setting the Clock Period. 1-14
Compiling and Writing the Gate-Level Netlist 1-14
Generating Summary Reports. 1-15
Synthesizing a Design With Multiple RTL Modules. 1-16
Analyzing and Elaborating Multiple RTL Modules 1-17
Setting the Clock Period. 1-18
Linking the .db Files 1-18
Compiling and Writing the Gate-Level Netlist 1-19
Synthesizing a Design With Integrated Behavioral and RTL Modules 1-19
Analyzing and Elaborating Multiple RTL and Behavioral Modules 1-21
Analyzing and Elaborating the RTL Modules 1-21
Analyzing and Elaborating the Behavioral Modules. 1-22
Analyzing and Elaborating the Integrated Design 1-23
Compiling and Writing the Gate-Level Netlist. 1-24

2. Creating SystemC Modules for RTL Synthesis

Defining Modules and Processes 2-2
Modules 2-2
Processes 2-3
Registering a Process 2-3
Triggering Execution of a Process 2-4
Reading and Writing Processes. 2-4

Types of Processes 2-4
Creating a Module 2-5
Module Header File 2-5
Module Syntax 2-6
Module Ports 2-7
Port Syntax 2-8
Port Data Types 2-8
Signals 2-9
Signal Syntax 2-10
Signal Data Types 2-10
Data Member Variables 2-11
Creating a Process in a Module 2-12
Defining the Sensitivity List 2-13
Defining a Level-Sensitive Process 2-13
Incomplete Sensitivity Lists 2-14
Defining an Edge-Sensitive Process 2-14
Limitations for Sensitivity Lists 2-16
Member Functions 2-16
Module Constructor 2-16
Implementing the Module 2-17
Reading and Writing Ports and Signals 2-17
Reading and Writing Bits of Ports and Signals 2-18
Signal and Port Assignments 2-18
Variable Assignment 2-20
Creating a Module With a Single SC_METHOD Process 2-22
Creating a Module With Multiple SC_METHOD Processes 2-24

Creating a Hierarchical RTL Module 2-28
The Basics of Hierarchical Module Creation 2-28
Hierarchical RTL Module Example 2-30
Creating an Integrated RTL and Behavioral Module 2-36

3. Using the Synthesizable Subset

Converting to a Synthesizable Subset. 3-2
Excluding Nonsynthesizable Code 3-2
SystemC and C++ Synthesizable Subsets 3-3
Nonsynthesizable SystemC Constructs 3-4
Nonsynthesizable C/C++ Constructs 3-5
Refining Data. 3-8
Synthesizable Data Types 3-8
Nonsynthesizable Data Types 3-9
Recommended Data Types for Synthesis 3-9
Using SystemC Data Types 3-12
Bit Vector Data Type Operators 3-12
Fixed-Precision and Arbitrary-Precision Data Type Operators 3-13
Using Enumerated Data Types 3-15
Using Aggregate Data Types 3-15
Using C++ Data Types 3-16
Data Members of a Module 3-17
Recommendations About Refinement 3-20

4. RTL Coding Guidelines

Register Inference 4-2
Flip-Flop Inference 4-2
Simple D Flip-Flop 4-2
D Flip-Flop With an Active-High Asynchronous Set or Reset 4-4
D Flip-Flop With an Active-Low Asynchronous Set or Reset 4-5
D Flip-Flop With Active-High Asynchronous Set and Reset 4-6
D Flip-Flop With Synchronous Set or Reset. 4-8
Inferring JK Flip-Flops 4-10
Inferring Toggle Flip-Flops 4-13
Latch Inference 4-16
Inferring a D Latch From an If Statement 4-16
Inferring An SR Latch 4-17
Avoiding Latch Inference 4-19
Inferring a Latch From a Switch Statement 4-21
Priority Encoding 4-25
Active-Low Set and Reset 4-27
Active-High Set and Reset 4-29
D Latch With an Asynchronous Set and Reset 4-30
D Latch With an Asynchronous Set 4-31
D Latch With an Asynchronous Reset 4-32
Understanding the Limitations of Register Inference 4-33
Multibit Inference 4-37
Inferring Multibit 4-38
Preventing Multibit Inference 4-40

Multiplexer Inference 4-42
Inferring Multiplexers From a Block of Code 4-42
Preventing Multiplexer Inference 4-45
Inferring a Multiplexer From a Specific Switch Statement 4-47
Understanding the Limitations of Multiplexer Inference 4-49
Three-State Inference 4-50
Simple Three-State Inference 4-50
Registered Three-State Drivers 4-52
Understanding the Limitations of Three-State Inference 4-55
Loops 4-56
Loop Unrolling Criteria 4-56
for Loop Counter 4-57
for Loop Comma Operator 4-58
Dead Loops 4-58
Infinite Loops 4-58
State Machines 4-59
State Machine With a Common Computation Process 4-61
State Machine With Separate Computation Processes 4-63
Moore State Machine 4-65
Defining a State Vector Variable 4-66

Appendix A. Compiler Directives

Synthesis Compiler Directives A-2
Line Label Compiler Directive A-3
Multibit Inference Compiler Directives A-3

Multiplexer Inference Compiler Directives A-4
Loop Unrolling Compiler Directive A-5
switch...case Compiler Directives A-5
Full Case A-5
Parallel Case A-6
State Vector Compiler Directive A-6
Enumerated Data Type Compiler Directive A-8
Synthesis Off and On A-9
C/C++ Compiler Directives A-9
C/C++ Line Label A-9
C/C++ Conditional Compilation A-10

Appendix B. Examples

Count Zeros Combinational Version B-2
Count Zeros Sequential Version B-6
Finite Impulse Response RTL Version B-11
Finite Impulse Response RTL and Behavioral Integrated Version B-18

Index

Figures

Figure 1-1 Behavioral Synthesis Compared to RTL Synthesis 1-3

Figure 1-2 SystemC Compiler Input and Output Flow for RTL Synthesis 1-7

Figure 1-3 Single RTL Module Command Flow 1-10

Figure 1-4 Command Flow for Multiple RTL Modules 1-16

Figure 1-5 Command Flow for Integrated RTL and Behavioral Modules 1-20

Figure 2-1 Module 2-3

Figure 2-2 Module Ports. 2-7

Figure 2-3 Processes and Signals 2-9

Figure 2-4 Signal Assignment Schematic 2-19

Figure 2-5 Variable Assignment Schematic 2-21

Figure 2-6 Count Zeros Combinational Module 2-22

Figure 2-7 Count Zeros Sequential Module 2-25

Figure 2-8 Finite Impulse Response RTL Modules 2-31

Figure 2-9 Finite Impulse Response RTL and Behavioral Integrated Modules 2-37

Figure 4-1 Inferred Positive-Edge-Triggered Flip-Flop 4-3
Figure 4-2 D Flip-Flop With an Active-High Asynchronous Reset. 4-5
Figure 4-3 D Flip-Flop With an Active-Low Asynchronous Reset. 4-6
Figure 4-4 Flip-Flop With Asynchronous Set and Reset 4-8
Figure 4-5 D Flip-Flop With Synchronous Reset 4-9
Figure 4-6 JK Flip-Flop. 4-11
Figure 4-7 JK Flip-Flop With Asynchronous Set and Reset. 4-13
Figure 4-8 Toggle Flip-Flop With Asynchronous Set 4-14
Figure 4-9 Toggle Flip-Flop With Asynchronous Reset 4-15
Figure 4-10 D Latch Inferred From an if Statement 4-17
Figure 4-11 SR Latch. 4-19
Figure 4-12 Avoiding Latch Inference by Adding Else Clause 4-20
Figure 4-13 Avoiding Latch Inference by a Default Value. 4-21
Figure 4-14 Latch Inference From a switch Statement. 4-22
Figure 4-15 Avoiding Latch Inference by Adding a Default Case to a switch Statement 4-23
Figure 4-16 Avoiding Latch Inference by Setting a Default Case Before a switch Statement 4-25
Figure 4-17 Using the full_case Compiler Directive With a switch Statement 4-27
Figure 4-18 Latch With Active-Low Set and Reset. 4-28
Figure 4-19 Latch With Active-High Set and Reset 4-30
Figure 4-20 Latch With Asynchronous Set and Reset 4-31
Figure 4-21 Latch With Asynchronous Set. 4-32

Figure 4-22 Latch With Asynchronous Reset. 4-33
Figure 4-23 Inferring a 2-Bit 4-to-1 Multiplexer. 4-39
Figure 4-24 Do Not Infer a 2-Bit 4-to-1 Multiplexer. 4-41
Figure 4-25 Inferring a Multiplexer for a Block 4-44
Figure 4-26 Block of Code Without Multiplexer Inference 4-46
Figure 4-27 Inferring a Multiplexer From a Specific switch Statement 4-48
Figure 4-28 Schematic of a Simple Three-State Driver 4-51
Figure 4-29 Three-State Driver With Gated Data. 4-52
Figure 4-30 Three-State Driver With Registered Enable 4-53
Figure 4-31 Three-State Driver Without Registered Enable. 4-54
Figure 4-32 Mealy State Machine 4-59
Figure 4-33 Finite State Machine State Diagram 4-60
Figure B-1 Count Zeros Combinational Module B-3
Figure B-2 Count Zeros Combinational Schematic B-5
Figure B-3 Count Zeros Sequential Module B-7
Figure B-4 Count Zeros Sequential Schematic B-10
Figure B-5 Finite Impulse Response RTL Modules B-11
Figure B-6 Finite Impulse Response RTL and Behavioral Integrated Modules B-18

Tables

Table 3-1 Nonsynthesizable SystemC Constructs for RTL Synthesis. 3-4

Table 3-2 Nonsynthesizable C/C++ Constructs 3-5

Table 3-3 Synthesizable Data Types. 3-10

Table 3-4 SystemC Bit Vector Data Type Operators. 3-12

Table 3-5 SystemC Integer Data Type Operators. 3-13

Table 4-1 Positive-Edge-Triggered JK Flip-Flop Truth Table 4-10

Table 4-2 Truth Table for the SR Latch (NAND Type) 4-18

Table A-1 SystemC Compiler Compiler Directives A-2

Examples

Example 2-1 Creating a Method Process in a Module 2-12
Example 2-2 Defining a Level-Sensitive Sensitivity List 2-13
Example 2-3 Incomplete Sensitivity List 2-14
Example 2-4 Defining an Edge-Sensitive Sensitivity List 2-15
Example 2-5 Module Implementation File 2-17
Example 2-6 Using Assignment and read() and write() Methods . . . 2-17
Example 2-7 Reading and Writing Bits of a Variable 2-18
Example 2-8 Signal Assignment 2-19
Example 2-9 Variable Assignment 2-21
Example 2-10 Count Zeros Combinational Version 2-22
Example 2-11 Count Zeros Sequential Version 2-25
Example 2-12 Hierarchical Module With Multiple RTL Modules 2-28
Example 2-13 Finite Impulse Response Top-Level Module 2-31
Example 2-14 Finite Impulse Response FSM Module 2-32
Example 2-15 Finite Impulse Response Data Module 2-34
Example 2-16 Finite Impulse Response Top-Level Integrated Module 2-37

Example 2-17 Finite Impulse Response Behavioral Module 2-38
Example 3-1 Excluding Simulation-Only Code. 3-3
Example 3-2 Enumerated Data Type 3-15
Example 3-3 Aggregate Data Type 3-15
Example 3-4 Implicit Bit Size Restriction 3-16
Example 3-5 Unknown Variable Bit Size 3-17
Example 3-6 Incorrect Use of a Data Member Variable for Interprocess Communication 3-17
Example 3-7 Correct Use of a Signal for Interprocess Communication 3-18
Example 4-1 Inferring a Positive-Edge-Triggered Flip-Flop 4-3
Example 4-2 D Flip-Flop With an Active-High Asynchronous Reset. 4-4
Example 4-3 D Flip-Flop With an Active-Low Asynchronous Reset. 4-5
Example 4-4 Flip-Flop With Asynchronous Set and Reset 4-7
Example 4-5 D Flip-Flop With Synchronous Reset 4-9
Example 4-6 JK Flip-Flop. 4-10
Example 4-7 JK Flip-Flop With Asynchronous Set and Reset 4-12
Example 4-8 Toggle Flip-Flop With Asynchronous Set 4-13
Example 4-9 Toggle Flip-Flop With Asynchronous Reset 4-14
Example 4-10 D Latch Inference Using an if Statement 4-16
Example 4-11 SR Latch 4-18
Example 4-12 Adding an Else Clause to Avoid Latch Inference 4-19
Example 4-13 Setting a Default Value to Avoid Latch Inference 4-20

Example 4-14 Latch Inference From a switch Statement 4-21
Example 4-15 Avoiding Latch Inference From a switch Statement 4-22
Example 4-16 Set a Default Value to Avoid Latch Inference From a switch Statement. 4-24
Example 4-17 Using the full_case Compiler Directive With a switch Statement 4-26
Example 4-18 Latch With Active-Low Set and Reset 4-28
Example 4-19 Latch With Active-High Set and Reset 4-29
Example 4-20 Latch With Asynchronous Set and Reset 4-30
Example 4-21 Latch With Asynchronous Set 4-31
Example 4-22 Latch With Asynchronous Reset 4-32
Example 4-23 Instantiating a Register That Cannot Be Inferred 4-34
Example 4-24 Inferring a 2-Bit 4-to-1 Multiplexer 4-38
Example 4-25 Do Not Infer a 2-Bit 4-to-1 Multiplexer 4-40
Example 4-26 Multiplexer Inference From a Block of Code 4-43
Example 4-27 No Multiplexer Inference From a Block of Code 4-45
Example 4-28 Multiplexer Inference From a Specific switch Statement 4-47
Example 4-29 Three-State Buffer Inference From a Block of Code 4-50
Example 4-30 Inferring One Three-State Driver 4-51
Example 4-31 Three-State Driver With Registered Enable 4-52
Example 4-32 Three-State Driver Without Registered Enable 4-53
Example 4-33 Implicit Bit Size Restriction 4-57
Example 4-34 Unknown Variable Bit Size 4-58

Example 4-35 Comma (,) Operator is Not Supported in a for Loop 4-58
Example 4-36 Dead Loop. 4-58
Example 4-37 Infinite Loops. 4-58
Example 4-38 State Machine With Common Computation Process 4-61
Example 4-39 State Machine With Separate Processes 4-63
Example 4-40 Moore State Machine 4-65
Example A-1 Using the state_vector Compiler Directive A-7
Example B-2 Count Zeros Combinational Version B-3
Example B-3 Count Zeros Combinational Design Command Script. B-5
Example B-4 Count Zeros Sequential Version B-7
Example B-5 Count Zeros Combinational Design Command Script. B-10
Example B-6 Finite Impulse Response Top-Level RTL Module B-11
Example B-7 Finite Impulse Response FSM Module B-12
Example B-8 Finite Impulse Response Data Module B-14
Example B-9 Finite Impulse Response RTL Hierarchical Module Command Script B-17
Example B-10 Finite Impulse Response Top-Level Integrated Module B-18
Example B-11 Finite Impulse Response Behavioral Module B-20
Example B-12 Finite Impulse Response Integrated Module Command Script B-22

Preface

This preface includes the following sections:

- What's New in This Release
- About This Guide
- Customer Support

What’s New in This Release

This section describes the new features, enhancements, and changes included in SystemC Compiler version 2001.08. Unless otherwise noted, you can find additional information about these changes later in this book.

New Features

SystemC Compiler version 2001.08 includes the following new features:

- Synthesis of an RTL SystemC description. Specify RTL synthesis by using the `compile_systemc` command with the `-rtl` and `-rtl_format` options.
- The `set_fpga` command defines the command flow is directed to an FPGA flow rather than to the default ASIC flow.

  For information about the FPGA command flow, see the *CoCentric SystemC Compiler Behavioral User Guide*.

Enhancements

SystemC Compiler version 2001.08 includes the following enhancements:

- A tutorial is provided as a quick start for learning about behavioral and RTL synthesis with SystemC Compiler. It is available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_tutorial.
- The example designs in the *CoCentric SystemC Compiler Behavioral Modeling Guide* and *CoCentric SystemC Compiler RTL User and Modeling Guide* are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_examples.

Known Limitations and Resolved STARs

Information about known problems and limitations, as well as about resolved Synopsys Technical Action Requests (STARs), is available in the *CoCentric SystemC Compiler Release Note* in SolvNet.

To see the *CoCentric SystemC Compiler Release Note,*

1. Go to the Synopsys Web page at http://www.synopsys.com and click SolvNet.
2. If prompted, enter your name and password.(If you do not have a Synopsys user name and password, click New Synopsys User Registration.)
3. Click Release Notes in the Main Navigation section, find the 2001.08 release, then open the *CoCentric SystemC Compiler Release Note*.

About This Guide

The *CoCentric SystemC Compiler RTL User and Modeling Guide* describes how to use SystemC Compiler for RTL synthesis. It also describes how to develop or refine a SystemC RTL model for synthesis with SystemC Compiler.

For information about SystemC, see the Open SystemC Community Web site at http://www.systemc.org.

Audience

The *CoCentric SystemC Compiler RTL User and Modeling Guide* is for designers with a basic knowledge of the SystemC Class Library, RTL design, and the C or C++ language and development environment.

Familiarity with one or more of the following Synopsys tools is helpful:

- Synopsys Design Compiler
- Synopsys HDL Compiler or VHDL Compiler
- Synopsys Scirocco VHDL Simulator
- Synopsys Verilog Compiled Simulator (VCS)

Related Publications

In addition to the *CoCentric SystemC Compiler RTL User and Modeling Guide*, see the following manuals:

- The *CoCentric SystemC Compiler Behavioral User Guide*, which provides information about how to synthesize a refined SystemC hardware behavioral module into an RTL or a gate-level netlist
- The *CoCentric SystemC Compiler Behavioral Modeling Guide*, which provides information about how to develop or refine a behavioral SystemC model for synthesis with SystemC Compiler
- The *CoCentric SystemC HDL CoSim User Guide*, which provides information about cosimulating a system with mixed SystemC and HDL modules
- The *SystemC User's Manual*, available from the Open SystemC Community Web site at http://www.systemc.org

For additional information about SystemC Compiler and other Synopsys products, see

- Synopsys Online Documentation (SOLD), which is included with the software
- Documentation on the Web, which is available through SolvNet on the Synopsys Web page at http://solvnet.synopsys.com. Solv-It article Synthesis-875.html provides an HTML index to the Solv-It articles related to synthesis with SystemC Compiler.
- The Synopsys MediaDocs Shop, from which you can order printed copies of Synopsys documents, at http://mediadocs.synopsys.com

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates command syntax.
`Courier italic`	Indicates a user-defined value in Synopsys syntax, such as *`object_name`*. (A user-defined value that is not Synopsys syntax, such as a user-defined value in a Verilog or VHDL statement, is indicated by regular text font italic.)
`Courier bold`	Indicates user input—text you type verbatim—in Synopsys syntax and examples. (User input that is not Synopsys syntax, such as a user name or password you enter in a GUI, is indicated by regular text font bold.)
[ ]	Denotes optional parameters, such as *`pin1`* [*`pin2 ... pinN`*]
|	Indicates a choice among alternatives, such as `low | medium | high` (This example indicates that you can enter one of three possible values for an option: low, medium, or high.)
_	Connects terms that are read as a single term by the system, such as `set_annotated_delay`
Control-c	Indicates a keyboard combination, such as holding down the Control key and pressing c.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center. Customer training is available through the Synopsys Customer Education Center.

Accessing SolvNet

SolvNet includes the Solv-It electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services including software downloads, documentation on the Web, and “Enter a Call With the Support Center.”

To access SolvNet,

1. Go to the SolvNet Web page at http://solvnet.synopsys.com.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, click New Synopsys User Registration.)

If you need help using SolvNet, click SolvNet Help in the column on the left side of the SolvNet Web page.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a call to your local support center from the Web by going to http://solvnet.synopsys.com (Synopsys user name and password required), then clicking “Enter a Call With the Support Center.”
- Send an e-mail message to support_center@synopsys.com.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at http://www.synopsys.com/support/support_ctr.

Training

For SystemC and SystemC Compiler training and private workshops, contact the Synopsys Customer Education Center in one of the following ways:

- Go to the Synopsys Web page at http://www.synopsys.com/services/education.
- Telephone (800) 793-3448.

A tutorial is provided as a quick start for learning about behavioral and RTL synthesis with SystemC Compiler. It is available in SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc/ccsc_tutorial.

Preface

xxx

1

Using SystemC Compiler for RTL Synthesis

SystemC Compiler synthesizes SystemC RTL modules or a design with integrated RTL and behavioral modules into a gate-level netlist. It can also synthesize a SystemC behavioral module into RTL or a gate-level netlist. After synthesis, you can this netlist as input to other Synopsys tools such as Design Compiler and Physical Compiler.

This chapter describes the RTL synthesis process and the commands you typically use, in the following sections:

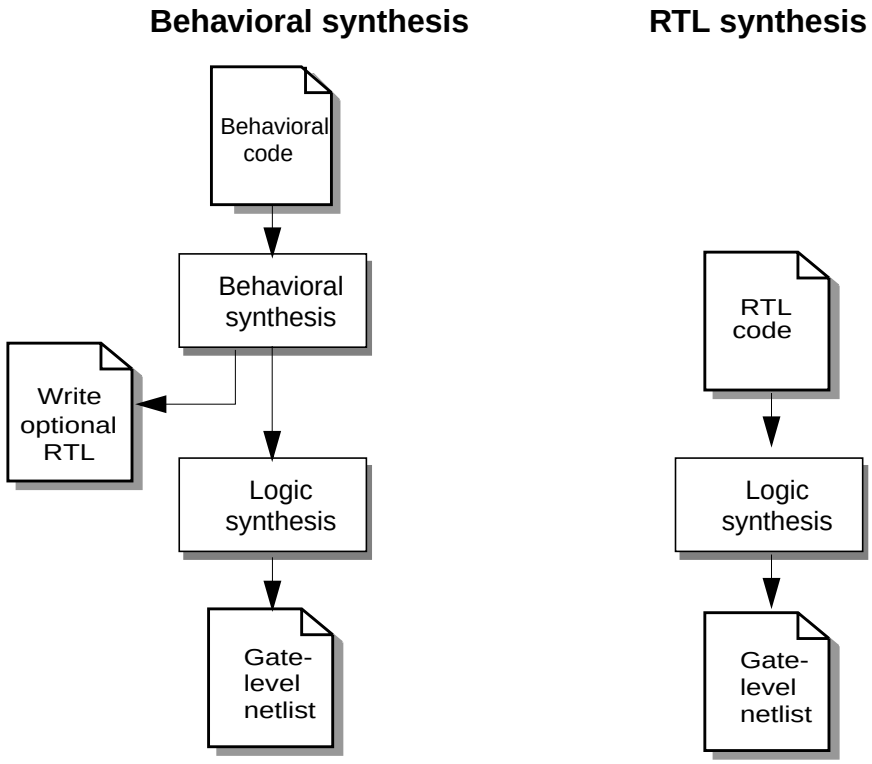
- Synthesis With SystemC Compiler
- Refinement Overview
- Inputs and Outputs for RTL Synthesis
- Synthesizing a Single RTL Module

- Synthesizing a Design With Multiple RTL Modules
- Synthesizing a Design With Integrated Behavioral and RTL Modules

Synthesis With SystemC Compiler

SystemC Compiler is a tool that can accept both behavioral and RTL SystemC descriptions and perform behavioral or RTL synthesis, as required, to create a gate-level netlist. You can also use SystemC Compiler to create an HDL description for simulation or to use with other HDL tools in your flow. Figure 1-1 shows the behavioral and RTL synthesis paths to gates with SystemC Compiler.

Figure 1-1 Behavioral Synthesis Compared to RTL Synthesis



Choosing the Right Abstraction for Synthesis

You can implement a hardware module by using RTL synthesis or behavioral-level synthesis. An RTL model describes registers in your design and the combinational logic between the registers. You specify the functionality of your system as an finite state machine (FSM) and a data path. Because register updates are tied to a clock, the model is cycle accurate, both at the interfaces and internally. Internal cycle accuracy that means you specify the clock cycle in which each operation is performed.

A behavioral model is an algorithmic description. Unlike with a pure software program, however, the I/O behavior of the model is described in a cycle accurate fashion. Therefore, wait statements are inserted into the algorithmic description to clearly delineate clock cycle boundaries and when I/O happens. Unlike RTL descriptions, the behavior is described algorithmically rather than in terms of an FSM and a data path.

Evaluate each design module by module, and consider each module's attributes, described in the following sections, to determine whether RTL or behavioral synthesis is applicable.

Identifying Attributes Suitable for RTL Synthesis

Look for the following design attributes when identifying a hardware module that is suitable for RTL synthesis with SystemC Compiler:

- It is easier to conceive the design as an FSM and a data path than as an algorithm—for example, a microprocessor.
- The design is very high-performance, and the designer, therefore, needs complete control over the architecture.

- The design contains complex memory such as SDRAM or RAMBUS.
- The design is asynchronous.

Identifying Attributes Suitable for Behavioral Synthesis

Look for the following design attributes when identifying a hardware module that is suitable for behavioral synthesis with SystemC Compiler:

- It is easier to conceive the design as an algorithm than as an FSM and a data path—for example, a fast Fourier transform, filter, inverse quantization, or digital signal processor.
- The design has a complex control flow—for example, a network processor.
- The design has memory accesses, and you need to synthesize access to synchronous memory.

For information about behavioral synthesis and modeling, see the *CoCentric SystemC Compiler Behavioral User Guide* and the *CoCentric SystemC Compiler Behavioral Modeling Guide*.

Refinement Overview

A pure C/C++ model of your hardware describes only what the hardware is intended to do, without providing information about the hardware structure or architecture. Starting with a C/C++ model, the goal of the first refinement stage is to create the hardware structure. To synthesize the hardware, you need to

- Define I/O ports for the hardware module
- Specify the internal structure as modules
- Specify the internal communication between the modules

For each block in the design, you start with a functional-level SystemC model and refine it into an RTL model for synthesis with SystemC Compiler. The stages of refining the high-level model into an RTL model for synthesis are

- Define the I/O in a cycle accurate fashion
- Separate the control logic and data path
- Determine the data-path architecture
- Define an explicit FSM for the control logic

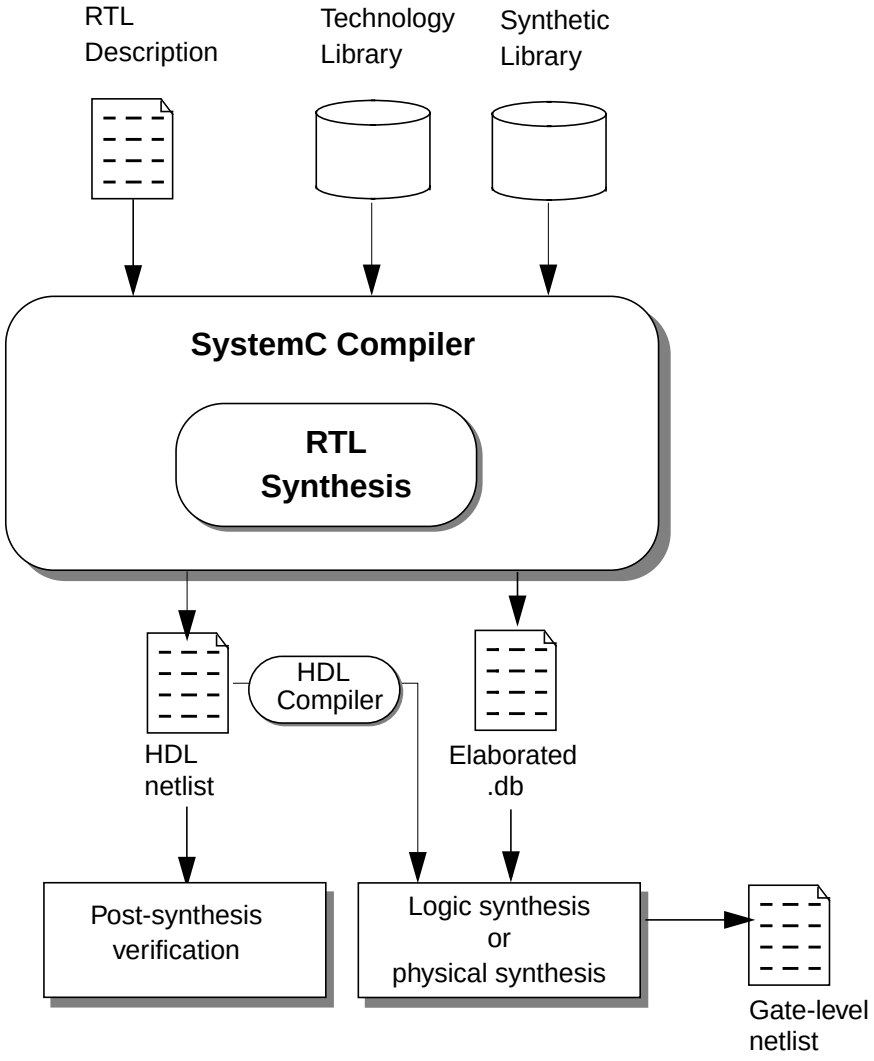
A high-level SystemC model can contain abstract ports, which are types that are not readily translated to hardware. For each abstract port, you need to define a port or a set of ports to replace each terminal of the abstract port, and replace all accesses to the abstract ports or terminals with accesses to the newly defined ports. For more information about abstract ports, see the *SystemC User's Guide*.

For further information about refinement, see "Recommendations About Refinement" on page 3-20 and "Refining Data" on page 3-8.

Inputs and Outputs for RTL Synthesis

SystemC Compiler requires a SystemC RTL description, a technology library, and a synthetic library. Figure 1-2 shows the flow into and out of SystemC Compiler.

Figure 1-2 SystemC Compiler Input and Output Flow for RTL Synthesis



RTL Description

Write the SystemC RTL description, using the SystemC Class Library according to the guidelines in Chapter 2, “Creating SystemC Modules for RTL Synthesis,” Chapter 3, “Using the Synthesizable Subset,” and Chapter 4, “RTL Coding Guidelines.”

The RTL description is independent of the technology. Using SystemC Compiler, you can change the target technology library without modifying the RTL description.

The example designs used in this manual are described in Appendix B, “Examples.” The files for these examples are available in the SystemC Compiler installation in the $SYNOPSYS/doc/syn/ccsc directory.

Technology Library

A technology library is provided by an ASIC vendor in Synopsys .db database format. It provides the area, timing, wire load models, and operating conditions. You provide the path to your chosen technology library for your design by defining the `target_library` variable in dc_shell.

Sample technology libraries are provided in the SystemC Compiler installation at $SYNOPSYS/libraries/syn.

Synthetic Library

The synthetic library is a technology-independent library of logic components such as adders and multipliers. SystemC Compiler maps your design operators to the synthetic library logical components. You provide the path to your chosen synthetic libraries for your design by defining the `synthetic_library` variable in dc_shell.

The DesignWare libraries are provided in the SystemC Compiler installation at $SYNOPSYS/libraries/syn. The synthetic libraries have names such as standard.sldb, dw01.sldb, and dw02.sldb. For information about the DesignWare libraries, see the DesignWare online documentation.

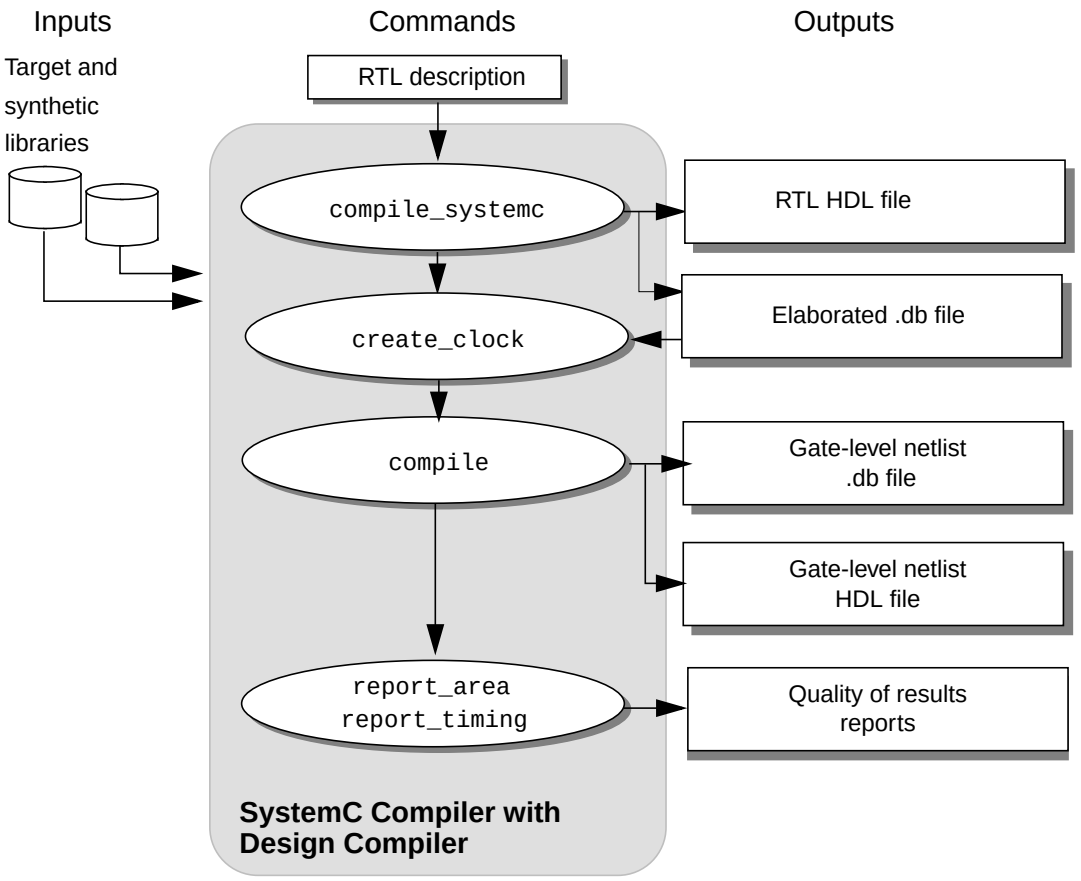
Outputs From SystemC Compiler

SystemC Compiler generates an elaborated .db file for input into the Design Compiler tool. It also generates RTL HDL files (currently only Verilog) that can be used in HDL-based flows.

Synthesizing a Single RTL Module

Figure 1-3 illustrates the primary commands you use to perform synthesis of a single RTL SystemC module with SystemC Compiler and compile the design into gates (using Design Compiler). The diagram also shows the inputs you provide and the outputs produced at various stages.

Figure 1-3 Single RTL Module Command Flow



The commands used in this chapter show the typical options you use. For a full description of a command and all its options, see the Synopsys online man pages. How to access and use man pages is described in Appendix A in the *CoCentric SystemC Compiler Behavioral User Guide*.

Starting SystemC Compiler

SystemC Compiler is integrated into Design Compiler. Enter the SystemC Compiler commands at the dc_shell prompt or use the `include` command to run a script that contains the commands. To start dc_shell, enter the following at a UNIX prompt:

```
unix% dc_shell
```

If this is the first time you are using SystemC Compiler, see Appendix A in the *CoCentric SystemC Compiler Behavioral User Guide* for information about setting up your environment, entering commands, and using scripts.

Analyzing and Elaborating a SystemC RTL Module

Use the `compile_systemc` command to read your SystemC source code and check it for compliance with synthesis policy, C++ syntax, and C++ semantics. If there are no errors, it produces an internal database (.db) ready for synthesis. This process is called analysis and elaboration.

The `compile_systemc` command, and the other SystemC Compiler commands, respond with a 1 if no errors were encountered or a 0 if an error was encountered. It also displays explanatory messages for errors and warnings.

The compile_systemc command does the following:

- Checks C++ syntax and semantics
- Replaces source code arithmetic operators with DesignWare components
- Performs optimizations such as constant propagating, constant folding, dead code elimination, and algebraic simplification

Creating an Elaborated .db File for Synthesis

To create an internal database (.db) of your SystemC RTL module for synthesis with Design Compiler,

- Analyze and elaborate your design, using the following command:

```
dc_shell> compile_systemc -rtl -rtl_format db
          design_name.cc
```

  where

  - The -rtl option means to synthesize in the RTL mode, which informs SystemC Compiler that the code contains SC_METHOD processes
  - The -rtl_format db option produces a database containing the design, which can be used by Design Compiler for logic synthesis

  When you use the -rtl and -rtl_format db options, the compile_systemc command sets the Design Compiler hdlin_enable_presto variable to true, sets the hdlin_unsigned_integers variable to false, and elaborates the design.

The compile_systemc command creates a ./rtl_work directory if it does not exist, but it does not write the .db file into this directory. To write the elaborated .db file into the ./rtl_work directory, use the write command with the -output option.

For example, to write the elaborated .db file as design_name_elab.db into the ./rtl_work directory, enter

```
dc_shell> write
     -hierarchy
     -output ./rtl_work/design_name_elab.db
```

Creating an RTL HDL Description

In certain cases, you might want to convert the SystemC RTL description into a Verilog or VHDL RTL description. (SystemC Compiler currently supports only Verilog output.)

To create a Verilog netlist of your SystemC RTL module, use the compile_systemc command with the -rtl option. The compile_systemc command reads your SystemC RTL source code, checks it for compliance with synthesis policy and semantics, and produces an RTL Verilog netlist.

1. To compile and create a Verilog netlist of your SystemC RTL design, execute the compile_systemc command with the following options:

   ```
   dc_shell> compile_systemc -rtl
           -rtl_format verilog
           design_name.cc
   ```

   When you execute the compile_systemc command with the -rtl and -rtl_format verilog options, it creates a Verilog .v file in the ./rtl_work directory named *design_name*.v. It creates the ./rtl_work directory if it does not exist.

2. Add the ./rtl_work directory to your `search_path` variable.

```
dc_shell> search_path = search_path + ./rtl_work
```

3. To analyze and elaborate the *design_name*.v file created in step 1 with HDL Compiler, enter

```
dc_shell> hdlin_enable_presto = "true"
dc_shell> hdlin_unsigned_integers = "false"
dc_shell> analyze -f verilog design_name.v
dc_shell> elaborate design_name
```

Setting the Clock Period

Use the `create_clock` command to set the clock period for the clock port in your design. The clock period uses the same unit defined in the target technology library. For example, to create a clock for the port in your design named *clk* with a period of 10 units, enter

```
dc_shell> create_clock clk -period 10
```

You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

Compiling and Writing the Gate-Level Netlist

Use the `compile` command to create the gate-level netlist. This command performs logic synthesis and optimization of the current design.

```
dc_shell> compile
     -map_effort medium
```

Use the following command to write the gate-level netlist in .db format:

```
dc_shell> write
     -hierarchy
     -output cmult_netlist.db
```

For verification at the gate level, write a Verilog or VHDL gate-level netlist file by entering the following command:

```
dc_shell> write
     -format verilog
     -hierarchy
     -output cmult_netlist.v
```

Or

```
dc_shell> write
     -format vhdl
     -hierarchy
     -output cmult_netlist.vhd
```

Generating Summary Reports

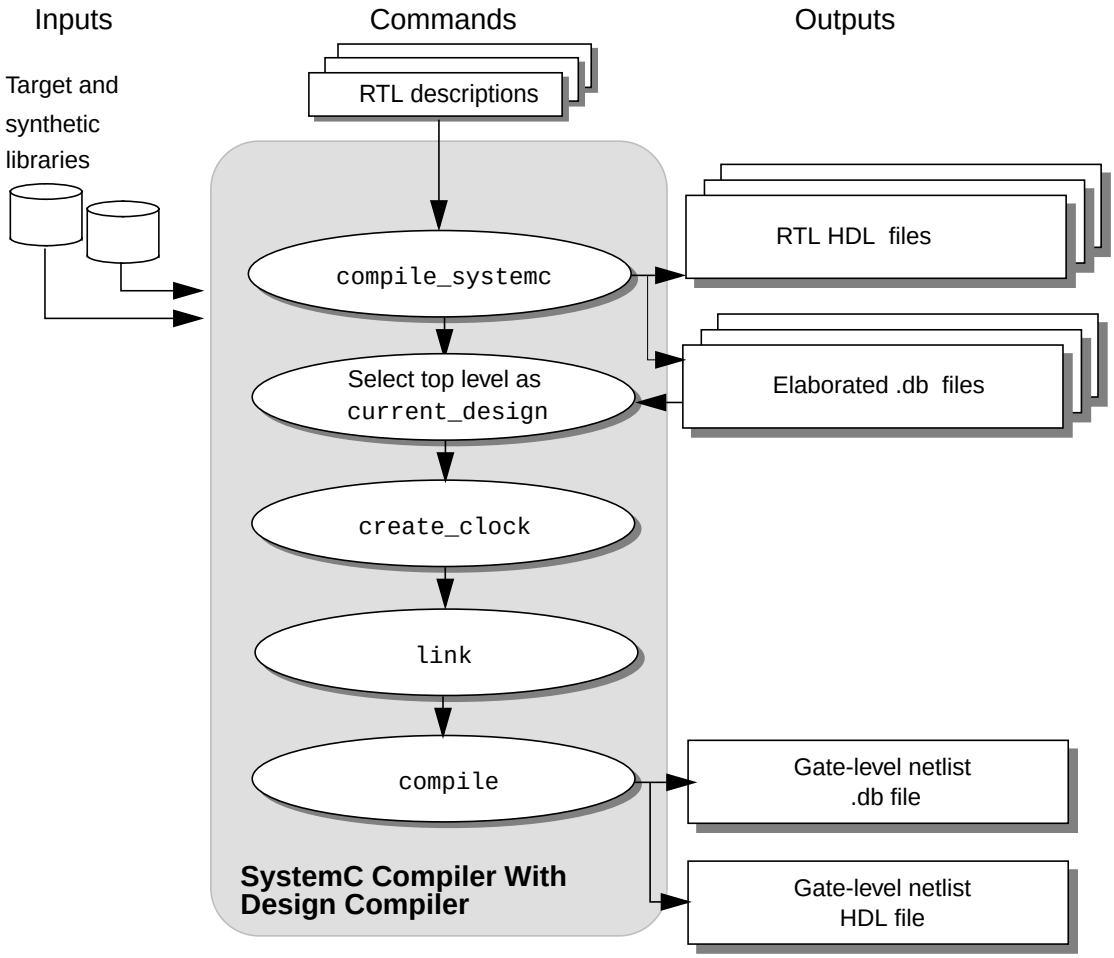
To generate summary reports of a design after it is compiled to gates, use one or both of the following commands:

```
dc_shell> report_area
dc_shell> report_timing
```

Synthesizing a Design With Multiple RTL Modules

Figure 1-4 illustrates the primary commands you use to perform synthesis of a design with multiple RTL SystemC modules and compile the design into gates. The diagram also shows the inputs you provide and the outputs SystemC Compiler can provide.

Figure 1-4 Command Flow for Multiple RTL Modules



Analyzing and Elaborating Multiple RTL Modules

If your design is hierarchical and contains multiple RTL modules, you need to use the `compile_systemc` command to elaborate each module separately, then use the `link` command to link the internal databases before compiling the design to gates.

To create internal database .db files of your SystemC RTL modules for synthesis,

- Analyze and elaborate each module in your design, using the following command:

```
dc_shell> compile_systemc -rtl -rtl_format db
          module1_name.cc
dc_shell> compile_systemc -rtl -rtl_format db
          module2_name.cc
dc_shell> compile_systemc -rtl -rtl_format db
          top_module_name.cc
```

  Use the `compile_systemc` command with the `-rtl` and `-rtl_format db` options to create a separate internal database for each module.

  To compile your entire design to gates, the current design in the Design Compiler database must be the top-level RTL module that instantiates all other RTL modules. To ensure that the current design in SystemC Compiler memory is the top-level RTL module, execute the `compile_systemc` command for the top RTL module last. Or you can use the `current_design` command to make the top-level module the current design. For example, to change the current design to my_design_abc, enter

```
dc_shell> current_design  my_design_abc
```

Setting the Clock Period

Use the `create_clock` command to set the clock period. Enter

```
dc_shell> create_clock -name clk -period 10
```

The clock period uses the same unit defined in the target technology library.

You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

Linking the .db Files

After analyzing and elaborating each module in your design and before compiling the design to gates, use the `link` command to connect all the library components and subdesigns your design references. Enter

```
dc_shell> link
```

The `link` command removes existing links before it starts the linking process. If you do not enter the `link` command to manually link the design references, the `compile` command performs linking but does not remove existing links.

Compiling and Writing the Gate-Level Netlist

Use the `compile` command to create the gate-level netlist of the hierarchical RTL design. This command performs logic synthesis and optimization of the current design.

```
dc_shell> compile
     -map_effort medium
```

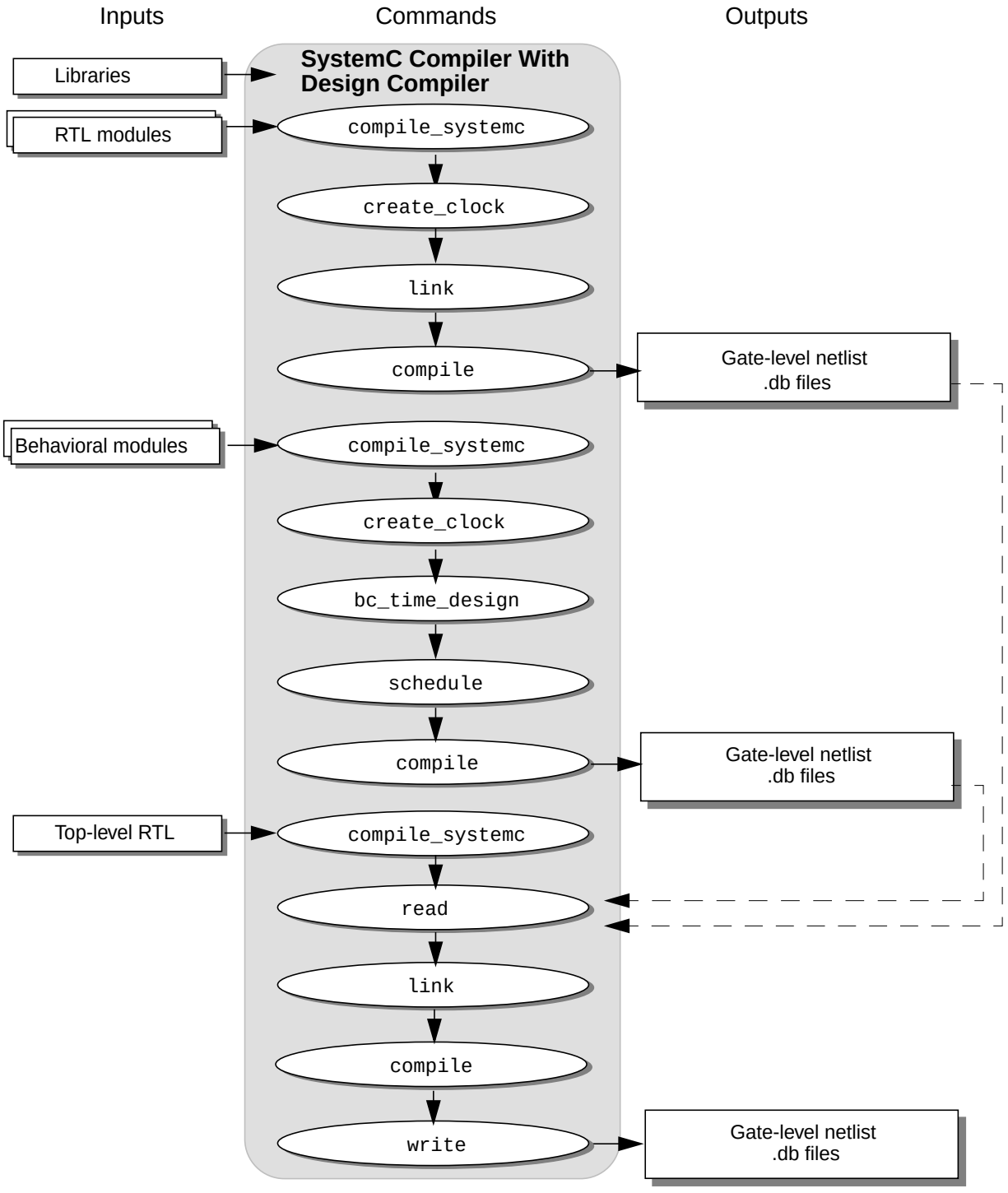
Use the following command to write the gate-level netlist in .db format:

```
dc_shell> write
     -hierarchy
     -output cmult_netlist.db
```

Synthesizing a Design With Integrated Behavioral and RTL Modules

Figure 1-5 illustrates the primary commands you use to perform synthesis of a design that integrates RTL and behavioral modules and compile the design into gates. The diagram also shows the inputs you provide and the outputs SystemC Compiler can provide.

Figure 1-5 Command Flow for Integrated RTL and Behavioral Modules



Analyzing and Elaborating Multiple RTL and Behavioral Modules

To perform synthesis of a design with integrated RTL and behavioral modules, synthesize the RTL and behavioral modules to gates before linking the integrated design.

Analyzing and Elaborating the RTL Modules

To create gate-level .db files of your SystemC RTL modules, perform the following steps:

1. Analyze and elaborate each RTL module in your design, using the following command:

   ```
   dc_shell> compile_systemc -rtl -rtl_format db
             module1_name.cc
   dc_shell> compile_systemc -rtl -rtl_format db
             module2_name.cc
   dc_shell> compile_systemc -rtl -rtl_format db
             top_module_name.cc
   ```

   Use the `compile_systemc` command with the `-rtl` and `-rtl_format db` options to create separate internal database files for each module.

2. Use the `create_clock` command to set the clock period. Enter

   ```
   dc_shell> create_clock clk -period 10
   ```

   You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

3. Link the .db files. Enter

```
dc_shell> link
```

   This links the individual modules and the library .db files for the design. You may not be able to link your entire design yet, because the behavioral modules are not yet synthesized.

4. Use the `compile` command to create the gate-level netlist of the hierarchical RTL design. Enter

```
dc_shell> compile
     -map_effort medium
```

5. (Optional) Use the `write` command to write the gate-level .db files for the RTL modules.

```
dc_shell> write
     -hierarchy
     -output rtl_top_gates.db
```

Analyzing and Elaborating the Behavioral Modules

A design can contain one or more behavioral modules. If your design has more than one behavioral module, instantiate multiple behavioral modules inside an RTL module. For details about the commands for performing behavioral synthesis, see the *CoCentric SystemC Compiler Behavioral User Guide*. For details about creating behavioral modules, see the *CoCentric SystemC Compiler Behavioral Modeling Guide*.

To create gate-level .db files of a SystemC behavioral module, perform the following steps:

1. Analyze and elaborate the behavioral module. Enter

```
dc_shell> compile_systemc  module1_name.cc
```

2. Use the `create_clock` command to set the clock period. Enter

```
dc_shell> create_clock clk -period 10
```

3. Use the `bc_time_design` command to estimate the timing and area. Enter

```
dc_shell> bc_time_design
```

4. Use the `schedule` command to schedule the design in the superstate-fixed or cycle-fixed I/O scheduling mode and allocate resources. Enter

```
dc_shell> schedule
     [-io_mode superstate_fixed | cycle_fixed]
```

5. Use the `compile` command to create the gate-level netlist of the behavioral module. Enter

```
dc_shell> compile
     -map_effort medium
```

6. (Optional) Use the `write` command to write the gate-level .db files of the behavioral modules.

```
dc_shell> write
     -hierarchy
     -output beh_gates.db
```

Analyzing and Elaborating the Integrated Design

To create the gate-level .db file for the top level of the integrated RTL and behavioral design, perform the following steps:

1. Analyze and elaborate the top-level RTL module that combines the RTL and behavioral modules. Enter

```
dc_shell> compile_systemc -rtl -rtl_format db  all_top.h
```

2. (Optional) If your RTL and behavioral modules are already in memory, you can skip this step. Otherwise, use the `read` command to read in the RTL and behavioral modules (.db files) that you previously compiled to gates. Enter

```
dc_shell> read saved_gates1.db
dc_shell> read saved_gates2.db
```

3. Use the `link` command to link the RTL, behavioral, and library .db into a single design.

```
dc_shell> link
```

Compiling and Writing the Gate-Level Netlist

Use the `compile` command to create the gate-level netlist of the integrated hierarchical RTL and behavioral design. This command performs logic synthesis and optimization of the current design.

You can set other optimization and design constraints before performing logic synthesis with the `compile` command. For information about optimization and design constraints for logic synthesis, see the Design Compiler documentation.

```
dc_shell> compile
     -map_effort medium
```

Use the following command to write the gate-level netlist of the entire design in .db format:

```
dc_shell> write
     -hierarchy
     -output cmult_netlist.db
```

2

Creating SystemC Modules for RTL Synthesis

This chapter explains the SystemC and C/C++ language elements that are important for RTL synthesis with SystemC Compiler. It contains the following sections:

- Defining Modules and Processes
- Creating a Module
- Creating a Module With a Single SC_METHOD Process
- Creating a Module With Multiple SC_METHOD Processes
- Creating a Hierarchical RTL Module
- Creating an Integrated RTL and Behavioral Module

Defining Modules and Processes

This modeling guide explains how to develop SystemC RTL modules for synthesis with SystemC Compiler. It assumes that you are knowledgeable about the C/C++ language and the SystemC Class Library available from the Open SystemC Community Web site at http://www.systemc.org.

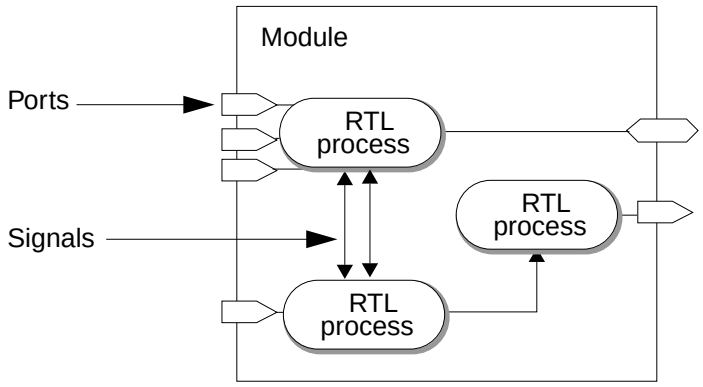
Modules

The basic building block in SystemC is the module. A SystemC module is a container in which processes and other modules are instantiated. For synthesis with SystemC Compiler, modules are either RTL or behavioral. A typical module can have

- Single or multiple RTL processes to specify combinational or sequential logic
- Single or multiple behavioral processes
- Multiple integrated behavioral or RTL modules to specify hierarchy
- One or more member functions that are called from within an instantiated process or module

Figure 2-1 shows a module with several RTL processes. The processes within a module are concurrent, and they execute whenever one of their sensitive inputs changes.

Figure 2-1 Module



Processes

SystemC provides processes to describe the parallel behavior of hardware systems. This means processes execute concurrently rather than sequentially like C++ functions. The code within a process, however, executes sequentially.

Registering a Process

Defining a process is similar to defining a C++ function. A process is declared as a member function of a module class and registered as a process in the module's constructor. Registering a process means that it is recognized as a SystemC process rather than as an ordinary member function.

You can register multiple different processes, but it is an error to register more than one instance of the same process. To create multiple instances of the same process, enclose the process in a module and instantiate the module multiple times.

Triggering Execution of a Process

You define a sensitivity list that identifies which input ports and signals trigger the execution of the code within a process. You can define level-sensitive inputs to specify combinational logic or edge-sensitive inputs to specify sequential logic.

Reading and Writing Processes

A process can read from and write to ports, internal signals, and internal variables.

Processes use signals to communicate with each other. One process can cause another process to execute by assigning a new value to a signal that interconnects them. Do not use data variables for communication between processes, because the processes execute in random order and it can cause nondeterminism (order dependencies) during simulation.

Types of Processes

SystemC provides three process types—SC_METHOD, SC_CTHREAD, and SC_THREAD—that execute whenever their sensitive inputs change. For simulation, you can use any of the process types. For RTL synthesis, you can use only the SC_METHOD process.

SC_METHOD Process. The SC_METHOD process is used for RTL synthesis with SystemC Compiler. It is sensitive to either changes in the signal values (level-sensitive) or to particular transitions (edges) of the signal (edge-sensitive) and executes when one of its sensitive inputs changes.

SC_CTHREAD Process. The SC_CTHREAD clocked thread process is used for behavioral synthesis with SystemC Compiler. It is sensitive to one edge of a clock signal. For information about creating behavioral processes, see the *CoCentric SystemC Compiler Behavioral Modeling Guide*.

Creating a Module

It is a recommended coding practice to describe a module by using a separate header file (*module_name*.h) and an implementation file (*module_name*.cpp or *module_name*.cc).

Module Header File

Each module header file contains

- Port declarations
- Internal signal variable declarations
- Internal data variable declarations
- Process declarations
- Member function declarations
- Module constructor

Module Syntax

Declare a module, using the syntax shown in bold in the following example:

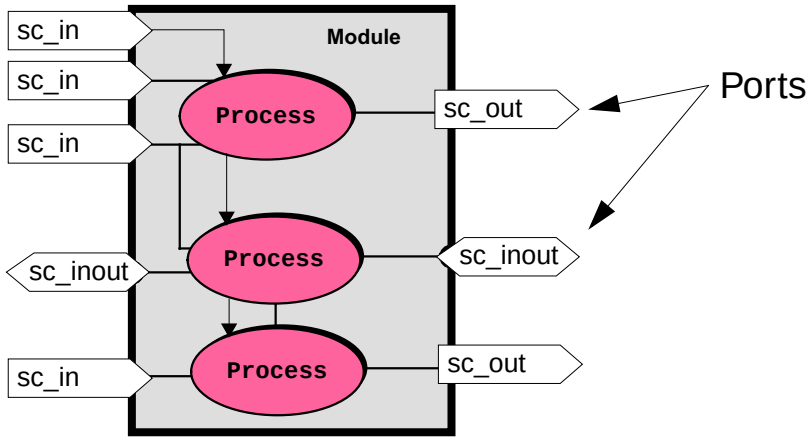
```
#include "systemc.h"
SC_MODULE (module_name) {
     //Module port declarations
     //Signal variable declarations
     //Data variable declarations
     //Member function declarations
     //Method process declarations

     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
     }
};
```

SC_MODULE and SC_CTOR are C++ macros defined in the SystemC Class library.

Module Ports

Each module has any number of ports (see Figure 2-2) that determine the direction of data into or out of the module.

Figure 2-2 Module Ports



A port is a data member of SC_MODULE. You can declare any number of sc_in, sc_out, and sc_inout ports. To read from an output port, declare it as an sc_inout rather than an sc_out port.

Note:

The `compile_systemc -rtl -rtl_format verilog` command converts an sc_inout port to a Verilog out port, not an inout port. You can read from and write to a Verilog out port. Verilog inout ports have restrictions for synthesis, as described in the *HDL Compiler (Presto Verilog) Reference Manual*.

Port Syntax

Declare ports by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     sc_in<port_data_type> port_name;
     sc_out<port_data_type> port_name;
     sc_inout<port_data_type> port_name;
     sc_in<port_data_type> port_name;


     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
     }
};
```

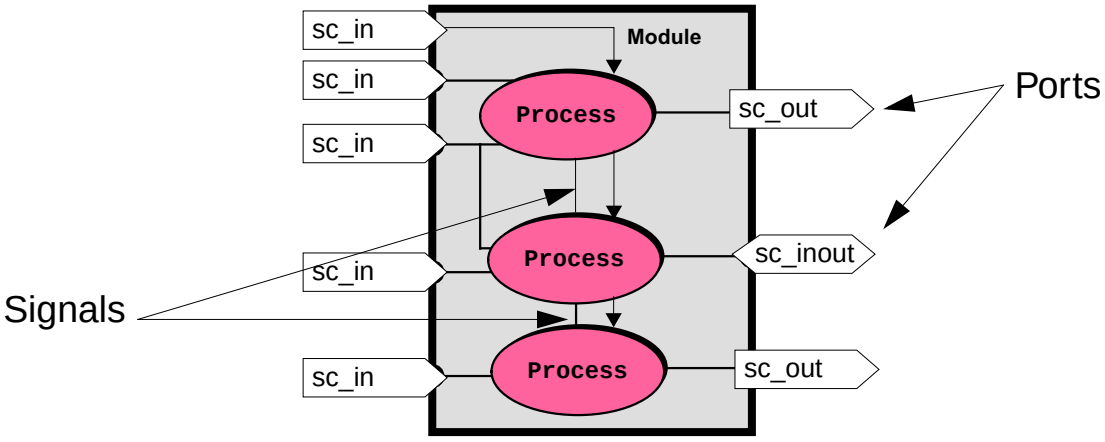
Port Data Types

Ports connect to signals and have a data type associated with them. For synthesis, declare each port as one of the synthesizable data types described in “Converting to a Synthesizable Subset” on page 3-2.

Signals

Modules use ports to communicate with other modules. In hierarchical modules, use signals to communicate between the ports of instantiated modules. Use internal signals for peer-to-peer communication between processes within the same module, as shown in Figure 2-3.

Figure 2-3 Processes and Signals



Signal Syntax

Declare signals by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     sc_in<port_type> port_name;
     sc_out<port_type> port_name;
     sc_in<port_type>port_name;

     //Internal signal variable declarations
     sc_signal<signal_type> signal_name;
     sc_signal<signal_type> signal1, signal2;

     //Data variable declarations
     //Process declarations
     //Member function declarations

     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
     }
};
```

Signal Data Types

A signal's bit-width is determined by its corresponding data type. Specify the data type as any of the synthesizable SystemC or C++ data types listed in "Converting to a Synthesizable Subset" on page 3-2. Signals and the ports they connect must have the same data types.

Data Member Variables

Inside a module, you can define data member variables of any synthesizable SystemC or C++ type. These variables can be used for internal storage in the module. Recommendations about using data member variables for synthesis are provided in “Data Members of a Module” on page 3-17. Declare internal data variables by using the syntax shown in bold in the following example:

```
SC_MODULE (module_name) {
     //Module port declarations
     sc_in<port_type> port_name;
     sc_out<port_type> port_name;
     sc_in port_name;

     //Internal signal variable declarations
     sc_signal<signal_type> signal_name;

     //Data member variable declarations
     int count_val;         //Internal counter
     sc_int<8> mem[1024];  //Array of sc_int

     //Process declarations
     //Member function declaration

     //Module constructor
     SC_CTOR (module_name) {
              //Register processes
              //Declare sensitivity list
     }
};
```

Note:

Do not use data variables for peer-to-peer communication in a module. This can cause presynthesis and postsynthesis simulation mismatches and nondeterminism (order dependency) in your design.

Creating a Process in a Module

SystemC processes are declared in the module body and registered as processes inside the constructor of the module, as shown in bold in Example 2-1.

You must declare a process with a return type of void and no arguments, which is also shown in bold in Example 2-1.

To register a function as an SC_METHOD process, you need to use the SC_METHOD macro that is defined in the SystemC class library. The SC_METHOD macro takes one argument, the name of the process.

Example 2-1 Creating a Method Process in a Module

```
SC_MODULE(my_module){
     // Ports
     sc_in<int> a;
     sc_in<bool> b;
     sc_out<int> x;
     sc_out<int> y;
     // Internal signals
     sc_signal<bool>c;
     sc_signal<int> d;
     // process declaration
    void my_method_proc();
        // module constructor
        SC_CTOR(my_module) {
            // register process
          SC_METHOD(my_method_proc);
            // Define the sensitivity list
     }
};
```

Defining the Sensitivity List

An SC_METHOD process reacts to a set of signals called its sensitivity list. You can use the sensitive(), sensitive_pos(), or sensitive_neg() functions or the sensitive, sensitive_pos, or sensitive_neg streams in the sensitivity declaration list.

Defining a Level-Sensitive Process

For combinational logic, define a sensitivity list that includes all input ports, inout ports, and signals used as inputs to the process. Use the sensitive method to define the level-sensitive inputs. Example 2-2 shows in bold a stream-type declaration and a function-type declaration. Specify any number of sensitive inputs for the stream-type declaration, and specify only one sensitive input for the function-type declaration. You can call the sensitive function multiple times with different inputs.

Example 2-2 Defining a Level-Sensitive Sensitivity List

```
SC_MODULE(my_module){
    // Ports
    sc_in<int> a;
    sc_in<bool> b;
    sc_out<int> x;
    sc_out<int> y;
    // Internal signals
    sc_signal<bool>c;
    sc_signal<int> d;
    sc_signal<int> e;
    // process declaration
    void my_method_proc();
    // module constructor
    SC_CTOR(my_module) {
        // register process
        SC_METHOD(my_method_proc);
        // declare level-sensitive sensitivity list
        sensitive << a << c << d; // Stream declaration
```

```
        sensitive(b);  //Function declaration
        sensitive(e);  //Function declaration
    }
};
```

Incomplete Sensitivity Lists

To eliminate the risk of presynthesis and postsynthesis simulation mismatches, include all the inputs to the combinational logic process in the sensitivity list of the method process. Example 2-3 shows an incomplete sensitivity list.

Example 2-3 Incomplete Sensitivity List

```
//method process
void comb_proc () {
    out_x = in_a & in_b & in_c;
}

SC_CTOR( comb_logic_complete ) {
  // Register method process
  SC_METHOD( comb_proc);
  sensitive << in_a << in_b;  // missing in_c
}
```

SystemC Compiler issues a warning if your sensitivity list is incomplete, but it proceeds to build a 3-input AND gate for the description in Example 2-3. When you simulate this description, however, out_x is not recalculated when in_c changes, because in_c is not in the sensitivity list. The simulated behavior, therefore, is not that of a 3-input AND gate.

Defining an Edge-Sensitive Process

For sequential logic, define a sensitivity list of the input ports and signals that trigger the process. Use the sensitive_pos, sensitive_neg, or both the sensitive_pos and sensitive_neg methods

to define the edge-sensitive inputs that trigger the process. Declare ports and the edge-sensitive inputs as type sc_in<bool>. You can define any number of sc_in<bool> inputs.

For edge-sensitive inputs, SystemC Compiler tests for the rising or falling edge of the signal. It infers flip-flops for variables that are assigned values in the process.

Define the sensitivity list by using either the function or the stream syntax. Example 2-4 shows in bold an example of a stream-type declaration for two inputs and a function-type declaration for the clock input.

Example 2-4 Defining an Edge-Sensitive Sensitivity List

```
SC_MODULE(my_module){
     // Ports
     sc_in<int> a;
     sc_in<bool> b;
     sc_in<bool> clock;
     sc_out<int> x;
     sc_out<int> y;
     sc_in<bool> reset;
     // Internal signals
     sc_signal<bool>c;
     sc_signal<int> d;
     // process declaration
     void my_method_proc();
     // module constructor
     SC_CTOR(my_module) {
          // register process
          SC_METHOD(my_method_proc);
          // declare sensitivity list
          sensitive_pos (clock); //Function delaration
          sensitive_neg << b << reset; // Stream declaration
     }
};
```

Limitations for Sensitivity Lists

When you define a sensitivity list, adhere to the following limitations:

- You cannot specify both edge-sensitive and level-sensitive inputs in the same process for synthesis.
- You cannot declare an sc_logic type for the clock or other edge-sensitive inputs. You can declare only an sc_in<bool> data type.

Member Functions

You can declare member functions in a module that are not processes. This type of member function is not registered as a process in the module's constructor. It can be called from a process. Member functions can contain any synthesizable C++ or SystemC statement allowed in a SC_METHOD process.

A member function that is not a process can return any synthesizable data type.

Module Constructor

For each module, you need to create a constructor, which is used to:

- Register processes
- Define a sensitivity list for an SC_METHOD process

For synthesis, other statements are not allowed in the constructor. Example 2-4 on page 2-15 shows a module constructor.

Implementing the Module

In the module implementation file, define the functionality of each SC_METHOD process and member function. Example 2-5 shows a minimal implementation file.

Example 2-5 Module Implementation File

```
#include "systemc.h"
#include "my_module.h"
void my_module::my_method_proc() {
   // describe process functionality as C++ code
}
```

Reading and Writing Ports and Signals

In the module implementation description, you can read from or write to a port or signal by using the read and write methods or by assignment.

When you read or write a port, as a recommended coding practice, use the read() and write() methods. Use the assignment operator for variables. Example 2-6 shows in bold how to use the read and write methods for ports and signals, and it shows assignment operators for variables.

Example 2-6 Using Assignment and read() and write() Methods

```
// read method
address = into.read();     // get address
// assignment
temp1 = address;                // save address
data_tmp = memory[address];  // get data from memory
// write method
outof.write(data_tmp);   // write out
// assignment
temp2 = data_tmp;              // save data_tmp
//...
```

Reading and Writing Bits of Ports and Signals

You read or write all bits of a port or signal. You cannot read or write the individual bits, regardless of the type. To do a bit-select on a port or signal, read the value into a temporary variable and do a bit-select on the temporary variable. Example 2-7 shows in bold how to read a signal into a temporary variable so you can read and write bits.

Example 2-7 Reading and Writing Bits of a Variable

```
//...
sc_signal <sc_int<8> > a;
sc_int<8> b;
bool c;
b = a.read();
c = b[0];

// c = a[0]; // Will not work in SystemC
```

Example 2-7 reads the value of signal a into temporary variable b, and bit 0 of b is written into variable c. You cannot read a bit from signal a, because this operation is not allowed in SystemC.

Signal and Port Assignments

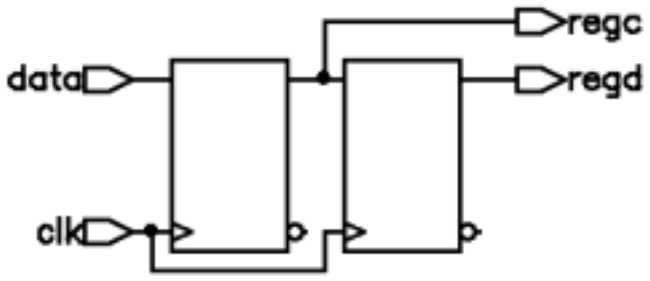
When you assign a value to a signal or port, the value on the right side is not transferred to the left side until the process ends. This means the signal values seen by other processes are not updated immediately, but deferred.

Example 2-8 shows a serial register implementation with signal assignment, and Figure 2-4 shows the resulting schematic.

Example 2-8 Signal Assignment

```
#include "systemc.h"

SC_MODULE(rtl_nb) {
  sc_in<bool> clk;
  sc_in<bool> data;
  sc_inout<bool> regc, regd;

  void reg_proc() {
    regc.write(data.read());
    regd.write(regc.read());
  }

  SC_CTOR(rtl_nb) {
    SC_METHOD(reg_proc);
    sensitive_pos << clk;
  }
};
```

Figure 2-4 Signal Assignment Schematic



Variable Assignment

When you assign a value to a variable, SystemC Compiler transfers the value on the right side immediately to the left side of the assignment statement.

Example 2-9 includes a variable assignment, where the implementation assigns the value of data to rega and regb, as the resulting schematic in Figure 2-5 indicates.

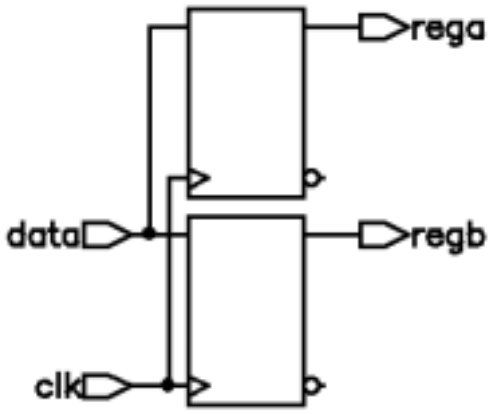
Note:

This example is only an illustration of variable assignment. You can write the same behavior more efficiently by removing the rega_v and regb_v variables and writing the ports directly.

Example 2-9 Variable Assignment

```
#include "systemc.h"

SC_MODULE(rtl_b) {
  sc_in<bool> clk;
  sc_in<bool> data;
  sc_out<bool> rega, regb;

  bool rega_v, regb_v;

  void reg_proc() {
    rega_v = data.read();
    regb_v = rega_v;
    rega.write(rega_v);
    regb.write(regb_v);
  }

  SC_CTOR(rtl_b) {
    SC_METHOD(reg_proc);
    sensitive_pos << clk;
  }
};
```

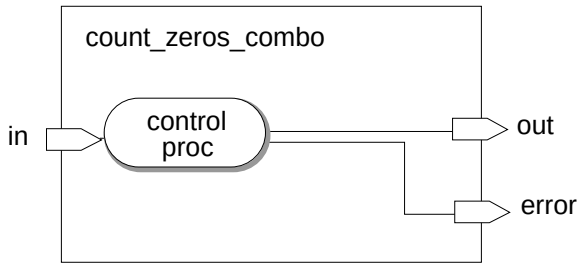
Figure 2-5 Variable Assignment Schematic



Creating a Module With a Single SC_METHOD Process

Example 2-10 is a complete example of an RTL description of a count zeros circuit that contains one SC_METHOD process, control_proc( ), and two member functions, legal( ) and zeros( ). The circuit determines in one cycle if an 8-bit value on the input port is valid (no more than one sequence of zeros) and how many zeros the value contains. The circuit produces two outputs, the number of zeros found and an error indication. Figure 2-6 illustrates the module and its ports. The design description and the complete set of files are shown in "Count Zeros Combinational Version" in Appendix B, and the files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc.

Figure 2-6 Count Zeros Combinational Module



Example 2-10 Count Zeros Combinational Version

```
/****count_zeros_comb.h file***/
#include "systemc.h"

SC_MODULE(count_zeros_comb) {
  sc_in<sc_uint<8> >  in;
  sc_out<sc_uint<4> >  out;
  sc_out<bool> error;

  bool legal(sc_uint<8> x);
  sc_uint<4> zeros(sc_uint<8> x);
  void control_proc();

  SC_CTOR(count_zeros_comb) {
```

```
    SC_METHOD(control_proc);
    sensitive << in;
  }
};

/****count_zeros_comb.cpp file****/
#include "count_zeros_comb.h"

void count_zeros_comb::control_proc() {
  sc_uint<4> tmp_out;
  bool is_legal = legal(in.read());
  error.write(! is_legal);
  is_legal ? tmp_out = zeros(in.read()) : tmp_out = 0;
  out.write(tmp_out);
}

bool count_zeros_comb::legal(sc_uint<8> x) {
  bool is_legal = 1;
  bool seenZero = 0;
  bool seenTrailing = 0;
  for (int i=0; i <=7; ++i) {
    if (seenTrailing && (x[i] == 0)) {
      is_legal = 0;
      break;
    } else if (seenZero && (x[i] == 1)) {
      seenTrailing = 1;
    } else if (x[i] == 0) {
      seenZero = 1;
    }
  }
  return is_legal;
}

sc_uint<4> count_zeros_comb::zeros(sc_uint<8> x) {
  int count = 0;
  for (int i=0; i <= 7; ++i) {
    if (x[i] == 0)
      ++count;
  }
  return count;
}
```

To synthesize a design similar to this example, use the commands in “Synthesizing a Single RTL Module” on page 1-10.

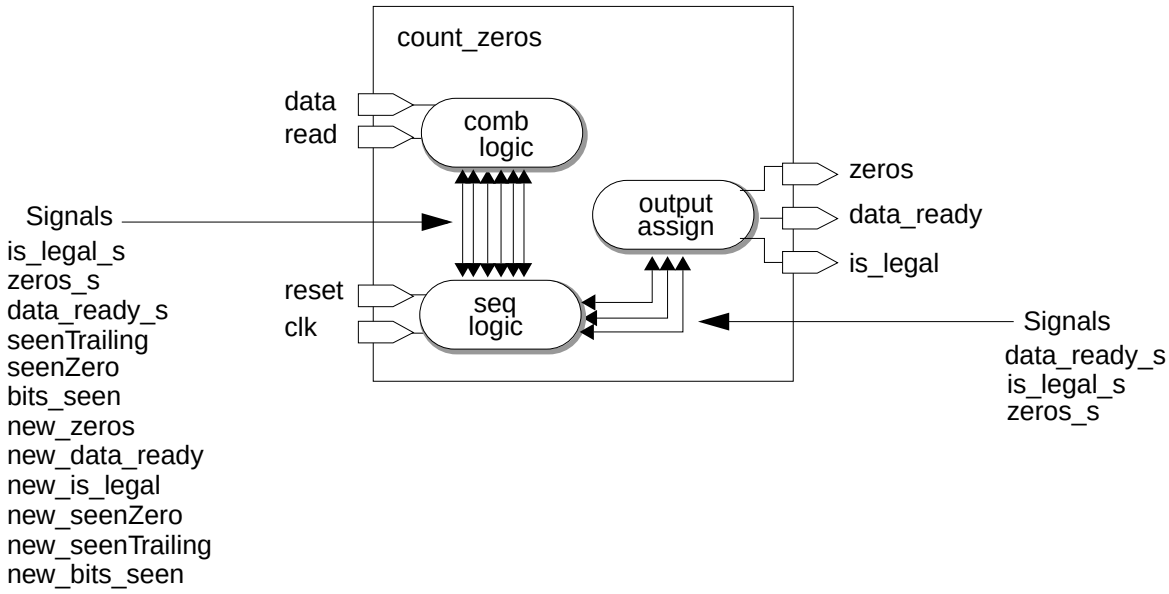
Creating a Module With Multiple SC_METHOD Processes

Example 2-11 on page 2-25 is a sequential description of the same count zeros circuit described in "Creating a Module With a Single SC_METHOD Process" on page 2-22. The complete set of files is shown in "Count Zeros Sequential Version" in Appendix B, and the files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc.

In this sequential version, there are three SC_METHOD processes and several signals for communication between the processes, as shown in Figure 2-7. The comb_logic( ) and output_assign( ) processes are level-sensitive, and the seq_logic( ) process is sensitive to the positive edge of the clk and reset inputs. The set_defaults( ) member function is called at the beginning of the comb_logic( ) process.

This example does not show typical simulation-specific code you might include for debugging purposes.

Figure 2-7 Count Zeros Sequential Module



Example 2-11 Count Zeros Sequential Version

```
/****count_zeros_seq.h file****/
#include "systemc.h"

#define ZEROS_WIDTH 4
#define MAX_BIT_READ 7

SC_MODULE(count_zeros_seq) {
  sc_in<bool> data, reset, read, clk;
  sc_out<bool> is_legal, data_ready;
  sc_out<sc_uint<ZEROS_WIDTH> > zeros;

  sc_signal<bool> new_data_ready, new_is_legal, new_seenZero, new_seenTrailing;
  sc_signal<bool> seenZero, seenTrailing;
  sc_signal<bool> is_legal_s, data_ready_s;
  sc_signal<sc_uint<ZEROS_WIDTH> > new_zeros, zeros_s;
  sc_signal<sc_uint<ZEROS_WIDTH - 1> > bits_seen, new_bits_seen;

  // Processes
  void comb_logic();
  void seq_logic();
  void assign_outputs();

  // Helper functions
  void set_defaults();
```

```
  SC_CTOR(count_zeros_seq) {
    SC_METHOD(comb_logic);
    sensitive << data << read << is_legal_s << data_ready_s;
    sensitive << seenTrailing << seenZero << zeros_s << bits_seen;

    SC_METHOD(seq_logic);
    sensitive_pos << clk << reset;

    SC_METHOD(assign_outputs);
    sensitive << is_legal_s << data_ready_s << zeros_s;
  }
};


/****count_zeros_seq.cpp file****/
#include "count_zeros_seq.h"

/*
 *  SC_METHOD: comb_logic()
 *    finds a singular run of zeros and counts them
 */
void count_zeros_seq::comb_logic() {
  set_defaults();
  if (read.read()) {
    if (seenTrailing && (data.read() == 0)) {
      new_is_legal = false;
      new_zeros = 0;
      new_data_ready = true;
    } else if (seenZero && (data.read() == 1)) {
      new_seenTrailing = true;
    } else if (data.read() == 0) {
      new_seenZero = true;
      new_zeros = zeros_s.read() + 1;
    }


    if (bits_seen.read() == MAX_BIT_READ)
      new_data_ready = true;
    else
      new_bits_seen = bits_seen.read() + 1;
  }
}

/*
 *  SC_METHOD: seq_logic()
 *    All registers have asynchronous resets
 */
```

```
void count_zeros_seq::seq_logic() {
  if (reset) {
    zeros_s = 0;
    bits_seen = 0;
    seenZero = false;
    seenTrailing = false;
    is_legal_s = true;
    data_ready_s = false;
  } else {
    zeros_s = new_zeros;
    bits_seen = new_bits_seen;
    seenZero = new_seenZero;
    seenTrailing = new_seenTrailing;
    is_legal_s = new_is_legal;
    data_ready_s = new_data_ready;
  }
}

/*
 *  SC_METHOD: assign_outputs()
 *  Zero time assignments of signals to their associated outputs
 */
void count_zeros_seq::assign_outputs() {
  zeros = zeros_s;
  is_legal = is_legal_s;
  data_ready = data_ready_s;
}

/*
 *   method: set_defaults()
 *     sets the default values of the new_* signals for the comb_logic
 *     process.
 */
void count_zeros_seq::set_defaults() {
  new_is_legal = is_legal_s;
  new_seenZero = seenZero;
  new_seenTrailing = seenTrailing;
  new_zeros = zeros_s;
  new_bits_seen = bits_seen;
  new_data_ready = data_ready_s;
}
```

To synthesize a design similar to this example, use the commands in “Synthesizing a Single RTL Module” on page 1-10.

Creating a Hierarchical RTL Module

You can create a hierarchical module with multiple instantiated modules. The lower-level modules can contain either SC_METHOD processes or a SC_CTHREAD behavioral process.

The Basics of Hierarchical Module Creation

To create a hierarchical module,

1. Create data members in the top-level module that are pointers to the instantiated modules.
2. Allocate the instantiated modules inside the constructor of the top-level module, giving each instance a unique name.
3. Bind the ports of the instantiated modules to the ports or signals of the top-level module. Use either binding by position or binding by name coding style, which is illustrated in bold in Example 2-12.

Example 2-12 shows the partial source code of two modules, fir_fsm and fir_data, instantiated within the fir_top module. The relevant code is highlighted in bold.

Example 2-12 Hierarchical Module With Multiple RTL Modules

```
/****fir_top.h****/
#include <systemc.h>
#include "fir_fsm.h"
#include "fir_data.h"

SC_MODULE(fir_top) {

  sc_in_clk         CLK;
  sc_in<bool>       RESET;
  sc_in<bool>       IN_VALID;
```

```
  sc_in<int>        SAMPLE;
  sc_out<bool>      OUTPUT_DATA_READY;
  sc_out<int>       RESULT;

  sc_signal<unsigned> state_out; //Communication between
                                 //two peer modules

  // Create data members - pointers to instantiated
  // modules
  fir_fsm  *fir_fsm1;
  fir_data *fir_data1;
  SC_CTOR(fir_top) {
     // Create new instance of fir_fsm module
      fir_fsm1 = new fir_fsm("FirFSM");

      // Binding by name
      fir_fsm1->clock(CLK);
      fir_fsm1->reset(RESET);
      fir_fsm1->in_valid(IN_VALID);
      fir_fsm1->state_out(state_out);

     // Binding by position alternative
     //fir_fsm1 (CLK, RESET, IN_VALID, state_out);

     // Create new instance
     // of fir_data module and bind by name
      fir_data1 = new fir_data("FirData");
      fir_data1->reset(RESET);
      fir_data1->state_out(state_out);
      fir_data1->sample(SAMPLE);
      fir_data1->result(RESULT);
      fir_data1->output_data_ready(OUTPUT_DATA_READY);
      fir_data1->clk(CLK);
       ...
    }
};

/****fir_fsm.h****/
SC_MODULE(fir_fsm) {

  sc_in<bool>      clock;
  sc_in<bool>      reset;
```

```
  sc_in<bool>      in_valid;
  sc_out<unsigned> state_out;
     ...

/****fir_data.h****/
SC_MODULE(fir_data) {

  sc_in<bool>      clk;
  sc_in<bool>      reset;
  sc_in<unsigned>  state_out;
  sc_in<int>       sample;
  sc_out<int>      result;
  sc_out<bool>     output_data_ready;
     ...
```

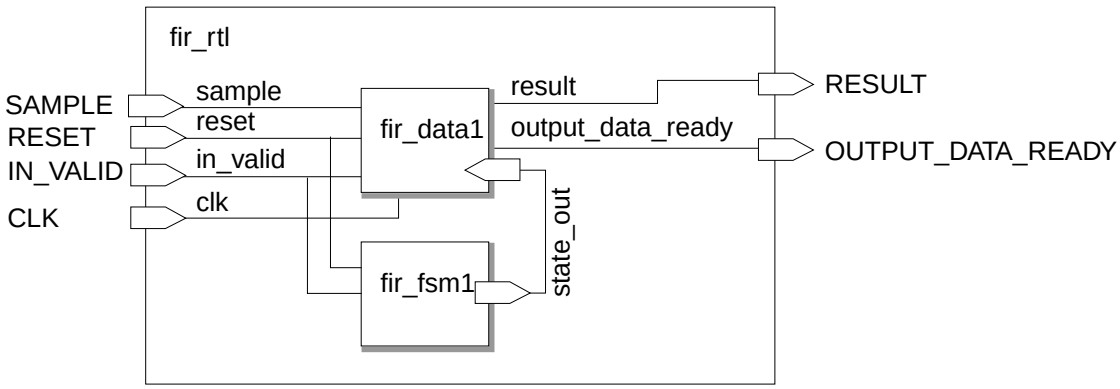
Hierarchical RTL Module Example

Example 2-13 on page 2-31 is a complete example of the hierarchical RTL finite impulse response filter description. This is a typical sequential logic description that separates the controlling FSM (Example 2-14 on page 2-32) and the data path (Example 2-15 on page 2-34) into two separate modules. Figure 2-8 illustrates the modules, the port binding, and their interconnecting signals. The design description and complete set of files are shown in "Finite Impulse Response RTL Version" in Appendix B, and the files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc.

In the top-level fir_rtl module, data member pointers to the fir_fsm and fir_data modules are declared, new instances of the two modules (fir_fsm1 and fir_data1) are created, and the port bindings are defined. A signal, state_out, is defined to connect the fir_fsm1 and fir_data1 state_out ports.

Coding guidelines for state machines are described in "State Machines" on page 4-59.

Figure 2-8 Finite Impulse Response RTL Modules



Example 2-13 Finite Impulse Response Top-Level Module

```
/****fir_rtl.h file****/
#include <systemc.h>
#include "fir_fsm.h"
#include "fir_data.h"

SC_MODULE(fir_rtl) {

  sc_in<bool>       clk;
  sc_in<bool>       reset;
  sc_in<bool>       in_valid;
  sc_in<int>        sample;
  sc_out<bool>      output_data_ready;
  sc_out<int>       result;

  sc_signal<unsigned> state_out;

  fir_fsm  *fir_fsm1;
  fir_data *fir_data1;

  SC_CTOR(fir_rtl) {

      fir_fsm1 = new fir_fsm("FirFSM");
      fir_fsm1->clock(clk);
      fir_fsm1->reset(reset);
```

```
      fir_fsm1->in_valid(in_valid);
      fir_fsm1->state_out(state_out);

      fir_data1 = new fir_data("FirData");
      fir_data1->state_out(state_out);
      fir_data1->sample(sample);
      fir_data1->clock(clk);
      fir_data1->result(result);
      fir_data1->output_data_ready(output_data_ready);
    }
};
```

Example 2-14 Finite Impulse Response FSM Module

```
/****fir_fsm.h file****/

SC_MODULE(fir_fsm) {

  sc_in<bool>      clock;
  sc_in<bool>      reset;
  sc_in<bool>      in_valid;
  sc_out<unsigned> state_out;

  // defining the states of the state machine
  enum {reset_s, first_s, second_s, third_s, output_s,
wait_s} state;

  SC_CTOR(fir_fsm)
    {
      SC_METHOD(entry);
      sensitive_pos(clock);
    };
  void entry();
};


/****fir_fsm.cpp file****/
#include <systemc.h>
#include "fir_fsm.h"
```

```
void fir_fsm::entry() {

  sc_uint<3> state_tmp;

  // reset behavior
  if(reset.read()==true) {
    state = reset_s;
  }
    // main state machine
    switch(state) {
    case reset_s:
      state = wait_s;
      state_tmp = 0;
      state_out.write(state_tmp);
      break;
    case first_s:
      state = second_s;
      state_tmp = 1;
      state_out.write(state_tmp);
      break;
    case second_s:
      state = third_s;
      state_tmp = 2;
      state_out.write(state_tmp);
      break;
    case third_s:
      state = output_s;
      state_tmp = 3;
      state_out.write(state_tmp);
      break;
    case output_s:
      state = wait_s;
      state_tmp = 4;
      state_out.write(state_tmp);
      break;

    default:
      if(in_valid.read()==true) {
      state = first_s;
      };
      state_tmp = 5;
      state_out.write(state_tmp);
```

```
      break;

    }
}
```

Example 2-15 Finite Impulse Response Data Module

```
/****fir_data.h file****/

C_MODULE(fir_data) {

  sc_in<unsigned>  state_out;
  sc_in<int>       sample;
  sc_out<int>      result;
  sc_out<bool>     output_data_ready;
  sc_in<bool>      clock;

  sc_int<19> acc;
  sc_int<8> shift[16];
  sc_int<9> coefs[16];

  SC_CTOR(fir_data)
    {
      SC_METHOD(entry);
      sensitive_pos(clock);
    };
  void entry();
};


/****fir_data.cpp file****/

#include <systemc.h>
#include "fir_data.h"

void fir_data::entry()
{
#include "fir_const_rtl.h"
  sc_int<8> sample_tmp;

  sc_uint<3> state = state_out.read();
```

```
switch (state) {
case 0:
  sample_tmp   = 0;
  acc = 0;
  for (int i=0; i<=15; i++) {
  shift[i] = 0;}
  result.write(0);
  output_data_ready.write(false);
  break;
case 1 :
  sample_tmp = sample.read();
  acc = sample_tmp*coefs[0];
  acc += shift[14]* coefs[15];
  acc += shift[13]*coefs[14];
  acc += shift[12]*coefs[13];
  acc += shift[11]*coefs[12];
  output_data_ready.write(false);
  break;
case 2 :
  acc += shift[10]*coefs[11];
  acc += shift[9]*coefs[10];
  acc += shift[8]*coefs[9];
  acc += shift[7]*coefs[8];
  output_data_ready.write(false);
  break;
case 3 :
  acc += shift[6]*coefs[7];
  acc += shift[5]*coefs[6];
  acc += shift[4]*coefs[5];
  acc += shift[3]*coefs[4];
  output_data_ready.write(false);
  break;
case 4 :
  acc += shift[2]*coefs[3];
  acc += shift[1]*coefs[2];
  acc += shift[0]*coefs[1];
  for(int i=14; i>=0; i--) {
    shift[i+1] = shift[i];
  };
  shift[0] = sample.read();
  result.write(acc);
```

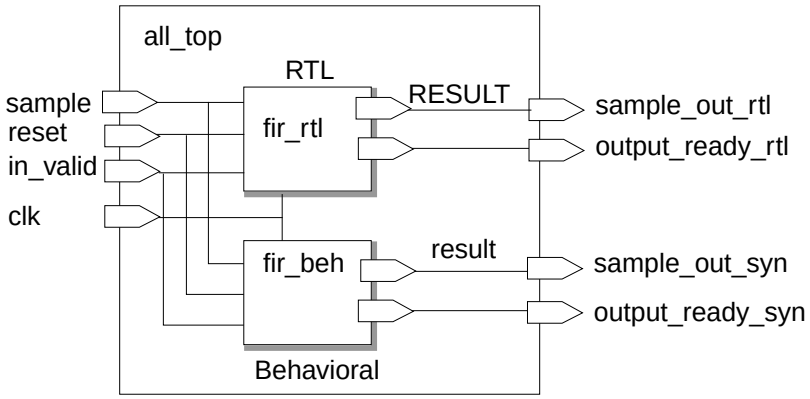
```
    output_data_ready.write(true);
    break;
  case 5 :
    // This state waits for valid input
    output_data_ready.write(false);
    break;
  default :
    output_data_ready.write(false);
    result.write(0);
  }
}
```

To synthesize a design similar to this example, use the commands in “Synthesizing a Design With Multiple RTL Modules” on page 1-16.

Creating an Integrated RTL and Behavioral Module

Creating an integrated RTL and behavioral module is similar to creating a hierarchical RTL module. Example 2-16 on page 2-37 shows an integrated module all_top that contains an instance of the hierarchical RTL fir_rtl module in Example 2-13 on page 2-31 and an instance of a behavioral version fir_beh of the finite impulse response filter shown in Example 2-17 on page 2-38. Figure 2-9 illustrates the modules and the port bindings. The design description and complete set of files are shown in “Finite Impulse Response RTL and Behavioral Integrated Version” in Appendix B, and the files are available in the SystemC Compiler installation in $SYNOPSYS/doc/syn/ccsc.

Figure 2-9 Finite Impulse Response RTL and Behavioral Integrated Modules



Example 2-16 Finite Impulse Response Top-Level Integrated Module

```
/****all_top.h file****/

#include <systemc.h>

#include "fir_rtl.h"
#include "fir_beh.h"

SC_MODULE(all_top) {

  sc_in<bool>       reset;
  sc_in<bool>       input_valid;

  sc_in<int>        sample;

  sc_out<int>       sample_out_rtl;
  sc_out<bool>      output_ready_rtl;
  sc_out<int>       sample_out_syn;
  sc_out<bool>      output_ready_syn;

  sc_in<bool>       clk1;

 // Instantiates RTL and behavioral models
  fir_rtl *fir_rtl1;
  fir_beh *fir_beh1;
```

```
  SC_CTOR(all_top) {

      fir_rtl1 = new fir_top("firTOP");
      fir_rtl1->reset(reset);
      fir_rtl1->in_valid(input_valid);
      fir_rtl1->sample(sample);
      fir_rtl1->result(sample_out_rtl);
      fir_rtl1->output_data_ready(output_ready_rtl);
      fir_rtl1->clk(clk1);

      fir_beh1 = new fir("FIR");
      fir_beh1->reset(reset);
      fir_beh1->input_valid(input_valid);
      fir_beh1->sample(sample);
      fir_beh1->result(sample_out_syn);
      fir_beh1->output_data_ready(output_ready_syn);
      fir_beh1->CLK(clk1);
      };
};
```

Example 2-17 Finite Impulse Response Behavioral Module

```
/****fir_beh.h file****/

C_MODULE(fir_beh) {

  sc_in<bool>  reset;
  sc_in<bool>  input_valid;
  sc_in<int>   sample;
  sc_out<bool> output_data_ready;
  sc_out<int>  result;
  sc_in_clk    CLK;

  SC_CTOR(fir_beh)
     {
      SC_CTHREAD(entry, CLK.pos());
      watching(reset.delayed() == true);
     }
```

```
  void entry();
};

/****fir_beh.cpp file****/

include <systemc.h>
#include "fir_beh.h"
#include "fir_const.h"

void fir_beh::entry() {

  sc_int<8>  sample_tmp;
  sc_int<17> pro;
  sc_int<19> acc;
  sc_int<8> shift[16];


  // reset watching
  for (int i=0; i<=15; i++)
    // synopsys unroll
    shift[i] = 0;
  result.write(0);
  output_data_ready.write(false);
  wait();

  // main functionality
  fir_loop:while(1) {

    output_data_ready.write(false);
    wait_until(input_valid.delayed() == true);

    sample_tmp = sample.read();
    acc = sample_tmp*coefs[0];

    for(int i=14; i>=0; i--) {
      // synopsys unroll
      acc += shift[i]*coefs[i+1];
    };


    for(int i=14; i>=0; i--) {
      // synopsys unroll
```

```
      shift[i+1] = shift[i];
    };
    shift[0] = sample_tmp;
    // write output values
    result.write(acc);
    output_data_ready.write(true);
    wait();
  };
}
```

To synthesize a design similar to this example, use the commands in “Synthesizing a Design With Integrated Behavioral and RTL Modules” on page 1-19.

3

Using the Synthesizable Subset

This chapter explains the subset of the SystemC and C/C++ language elements and data types that are used for RTL synthesis with SystemC Compiler. It contains the following sections:

- Converting to a Synthesizable Subset
- Refining Data
- Recommendations About Refinement

Converting to a Synthesizable Subset

To prepare for synthesis, you need to convert all nonsynthesizable code into synthesizable code. This is required only for functionality that is to be synthesized.

Although you can use any SystemC class or C++ construct for simulation and other stages of the design process, only a subset of the language can be used for synthesis. SystemC Compiler does not recognize nonsynthesizable constructs, and it displays an error message if it encounters any of these constructs in your code. You can use #ifdef and #endif to comment out code that is needed only for simulation. For example, you can exclude trace and print statements with these compiler directives.

Excluding Nonsynthesizable Code

SystemC Compiler provides compiler directives you can use in your code

- To include synthesis-specific directives
- To exclude or comment out nonsynthesizable and simulation-specific code so it does not interfere with synthesis

You can isolate nonsynthesizable code or simulation-specific code with a compiler directive, either the C language #ifdef and #endif (recommended) or a comment starting with the word `synopsys` or `snps` and `synthesis_off`. Example 3-1 shows compiler directives in bold that exclude simulation code for synthesis.

Example 3-1 Excluding Simulation-Only Code

```
//C directive  (recommended style)
#ifdef SIM
...//Simulation-only code
#endif

//SystemC Compiler directive
//(using #ifdef instead is recommended)
/* synopsys synthesis_off */
... //Simulation-only code
/* snps synthesis_on */
```

For this example, if the symbol SIM is defined, the tracing function is compiled with the intent of doing a simulation. If it is not defined, the tracing function is not included in the code translated by SystemC Compiler and is therefore not simulated.

You can define the SIM symbol with a #define directive, or you can provide it in the compiler command line for simulation purposes.

SystemC and C++ Synthesizable Subsets

The synthesizable subsets of SystemC and C++ are provided in the sections that follow. Wherever possible, a recommended corrective action is indicated for converting nonsynthesizable constructs into synthesizable constructs. For many nonsynthesizable constructs, there is no obvious recommendation for converting them into synthesizable constructs or there are numerous ways to convert them. In such cases, a recommended corrective action is not indicated. Familiarize yourself with the synthesizable subset and use it as much as possible in your pure C/C++ or high-level SystemC models to minimize the refinement effort for synthesis.

You can use any SystemC or C++ construct for a testbench. You do not need to restrict your code to the synthesizable subset in the testbench.

Nonsynthesizable SystemC Constructs

SystemC Compiler does not support the SystemC constructs listed in Table 3-1 for RTL synthesis.

Table 3-1 Nonsynthesizable SystemC Constructs for RTL Synthesis

Category	Construct	Comment	Corrective action
Thread process	SC_THREAD	Used for modeling a testbench but not supported for synthesis.	
CTHREAD process	SC_CTHREAD	Used for simulation and modeling at the behavioral level.	
Main function	sc_main()	Used for simulation.	
Clock generators	sc_start()	Used for simulation.	Use only in sc_main( ).
Global watching	watching()	Not supported for RTL synthesis	
Local watching	W_BEGIN, W_END, W_DO, W_ESCAPE	Not supported.	
Tracing	sc_trace, sc_create* trace_file	Creates waveforms of signals, channels, and variables for simulation.	Comment out for synthesis.

Nonsynthesizable C/C++ Constructs

SystemC Compiler does not support the C and C++ constructs listed in Table 3-2 for RTL synthesis.

Table 3-2 Nonsynthesizable C/C++ Constructs

Category	Construct	Comment	Corrective action
Local class declaration		Not allowed.	Replace.
Nested class declaration		Not allowed.	Replace.
Derived class		Only SystemC modules and processes are supported.	Replace.
Dynamic storage allocation	malloc(), free(), new, new[], delete[]	malloc(), free(), new, new[], delete, and delete[] not supported.	Use static memory allocation.
Exception handling	try, catch, throw	Not allowed.	Comment out.
Recursive function call		Not allowed.	Replace with iteration.
Function overloading		Not allowed (except the classes overloaded by SystemC).	Replace with unique function calls.
C++ built-in functions		Math library, I/O library, file I/O, and similar built-in C++ functions not allowed.	Replace with synthesizable functions or remove.
Virtual function		Not allowed.	Replace with a nonvirtual function.
Inheritance		Not allowed.	Create an independent SC_MODULE.
Multiple inheritance		Not allowed.	Create independent modules.

Table 3-2 Nonsynthesizable C/C++ Constructs (Continued)

Category	Construct	Comment	Corrective action
Member access control specifiers	public, protected, private, friend	Allowed in code but ignored for synthesis. All member access is public.	
Accessing struct members with the (->) operator	-> operator	Not allowed, except for module instantiation.	Replace with access using the period (.) operator.
Static member		Not allowed.	Replace with non-static member variable.
Dereference operator	* and & operators	Not allowed.	Replace dereferencing with array accessing.
for loop comma operator	, operator	The comma operator is not allowed in a for loop definition.	Remove the comma operators.
Operator overloading		Not allowed (except the classes overloaded by SystemC).	Replace overloading with unique function calls.
Operator, sizeof	sizeof	Not allowed.	Determine size statically for use in synthesis.
Pointer	*	Pointers are allowed only in hierarchical modules to instantiate other modules.	Replace all other pointers with access to array elements or individual elements.
Pointer type conversions		Not allowed.	Do not use pointers. Use explicit variable reference.
this pointer	this	Not allowed.	Replace.
Reference, C++	&	Allowed only for passing parameters to functions.	Replace in all other cases.

Table 3-2 Nonsynthesizable C/C++ Constructs (Continued)

Category	Construct	Comment	Corrective action
Reference conversion		Reference conversion is supported for implicit conversion of signals only.	Replace in all other cases.
User-defined template class		Only SystemC templates classes such as sc_int<> are supported.	Replace.
Type casting at runtime		Not allowed.	Replace.
Type identification at runtime		Not allowed.	Replace.
Explicit user-defined type conversion		The C++ built-in types and SystemC types are supported for explicit conversion.	Replace in all other cases.
Unconditional branching	goto	Not allowed.	Replace.
Unions		Not allowed.	Replace with structs.
Global variable		Not supported for synthesis.	Replace with local variables.
Member variable		Member variables accessed by two or more SC_METHOD processes are not supported. However, access to member variables by only one process is supported.	Use signals instead of variables for communication between processes.
Volatile variable		Not allowed.	Use only nonvolatile variables.

Refining Data

A pure C/C++ model or a high-level SystemC model typically uses native C++ types or aggregates (structures) of such types. Native C++ types such as int, char, bool, and long have fixed, platform-dependent widths, which are often not the correct width for efficient hardware. For example, you might need only a 6-bit integer for a particular operation, instead of the native C++ 32-bit integer. In addition, C++ does not support four-valued logic vectors, operations such as concatenation, and other features that are needed to efficiently describe hardware operations.

SystemC provides a set of limited-precision and arbitrary-precision data types that allows you to create integers, bit vectors, and logic vectors of any length. SystemC also supports all common operations on these data types.

To refine a SystemC model for synthesis, you need to evaluate all variable declarations, formal parameters, and return types of all functions to determine the appropriate data type and the appropriate widths of each data type. The following sections provide recommendations about the appropriate data type to use and when. Selecting the data widths is a design decision, and it is typically a tradeoff between the cost of hardware and the required precision. This decision is, therefore, left to you.

Synthesizable Data Types

C++ is a strongly typed language. Every constant, port, signal, variable, function return type, and parameter is declared as a data type, such as bool or sc_int<*n*>, and can hold or return a value of that type. Therefore, it is important that you use the correct data types in expressions.

Nonsynthesizable Data Types

All SystemC and C++ data types can be used for RTL synthesis, except the following types:

- Floating-point types such as float and double
- Fixed-point types sc_fixed, sc_ufixed, sc_fix, and sc_ufix
- Access types such as pointers
- File types such as FILE
- I/O streams such as stdout and cout

Recommended Data Types for Synthesis

For best synthesis, use appropriate data types and bit-widths so SystemC Compiler does not build unnecessary hardware.

The following are some general recommendations about data type selections:

- For a single-bit variable, use the native C++ type bool.
- For variables with a width of 64 bits or less, use sc_int or sc_uint data types. Use sc_uint for all logic and unsigned arithmetic operations. Use sc_int for signed arithmetic operations as well as for logic operations. These types produce the fastest simulation runtimes of the SystemC types.
- For variables larger than 64 bits, use sc_bigint or sc_biguint if you want to do arithmetic operations with these variables.

- Use sc_logic or sc_lv only when you need to model three-state signals or buses. When you use these data types, avoid comparison with X and Z values in your synthesizable code, because such comparisons are not synthesizable.
- Use native C++ integer types for loop counters.
- Use the native C++ data types with caution, because their size is platform-dependent. For example, on most platforms, a char is 8 bits wide, a short is 16 bits wide, and both an int and long are 32 bits wide. An int, however, can be 16, 32, or 64 bits wide.

To restrict bit size for synthesis, use the recommended SystemC data types summarized in Table 3-3 in place of the equivalent C++ native type. For example, change an int type to an sc_int<*n*> type.

Table 3-3 Synthesizable Data Types

SystemC and C++ type	Description
sc_bit	A single-bit true or false value. Supported but not recommended. Use the bool data type.
sc_bv<*n*>	Arbitrary-length bit vector. Use sc_uint<*n*> when possible.
sc_logic	A single-bit 0, 1, X, or Z.
sc_lv<*n*>	Arbitrary-length logic vector.
sc_int<*n*>	Fixed-precision integers with a maximum size of 64 bits and 64 bits of precision during operations.
sc_uint<*n*>	Fixed-precision integers with a maximum size of 64 bits and 64 bits of precision during operations, unsigned.
sc_bigint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits and unlimited precision.
sc_biguint<*n*>	Arbitrary-precision integers recommended for sizes over 64 bits and unlimited precision, unsigned.

Table 3-3 Synthesizable Data Types

SystemC and C++ type	Description
bool	A single-bit true or false value.
int	A signed integer, typically 32 or 64 bits, depending on the platform.
unsigned int	An unsigned integer, typically 32 or 64 bits, depending on the platform.
long	A signed integer, typically 32 bits or longer, depending on the platform.
unsigned long	An unsigned integer, typically 32 bits or longer, depending on the platform.
char	8-bit signed character, platform-dependent.
unsigned char	8-bit unsigned character, platform-dependent.
short	A signed short integer, typically 16 bits, depending on the platform.
unsigned short	An unsigned short integer, typically 16 bits, depending on the platform.
struct	A user-defined aggregate of synthesizable data types.
enum	A user-defined enumerated data type associated with an integer constant.

Using SystemC Data Types

Use the SystemC data type operators to access individual bits of a value.

Bit Vector Data Type Operators

Table 3-4 lists the operators available for the SystemC sc_bv data type.

Table 3-4 SystemC Bit Vector Data Type Operators

Operators
Bitwise &(and), |(or), ^(xor), and ~(not)
Bitwise <<(shift left) and >>(shift right)
Assignment =, &=, |=, and ^=
Equality ==, !=
Bit selection [x]
Part selection range (x-y)
Concatenation (x,y)
Reduction: and_reduce( ), or_reduce( ), and xor_reduce( )
Type conversion: to_uint( ) and to_int( )

Fixed-Precision and Arbitrary-Precision Data Type Operators

Table 3-5 lists the operators available for the SystemC sc_int and sc_uint fixed-precision and sc_bigint and sc_biguint arbitrary-precision integer data types.

Table 3-5 SystemC Integer Data Type Operators

Operators
Bitwise &(and), |(or), ^(xor), and ~(not)
Bitwise <<(shift left) and >>(shift right)
Assignment =, &=, |=, ^=, +=, -=, *=, /=, and %=
Equality ==, !=
Relational <, <=, >, and >=
Autoincrement ++ and autodecrement --
Bit selection [x]
Part selection range (x-y)
Concatenation (x,y)
Type conversion: to_uint( ) and to_int( )

Note:

The reduction and_reduce(), or_reduce(), and xor_reduce() operators are not available for the fixed- and arbitrary-precision data types.

Concatenating Variables. Variables must be the same SystemC data type to use the concatenation operation (,). SystemC Compiler reports an error if your code concatenates variables of different SystemC data types. For example, the following code produces an error, because the data type of the parity variable is not the same as the data types of a and b:

```
...
sc_uint<16> a = 0;
sc_uint<15> b = 0;
bool parity;
sc_uint<32> c = 0;
...
c = (a, b, parity);
...
```

To correct this coding error, the data types of variables parity, b, and c must be the same. For example,

```
...
sc_uint<16> a = 0;
sc_uint<15> b = 0;
sc_uint<1> parity;
sc_uint<32> c = 0;
...
c = (a, b, parity);
...
```

The equality operation (=) has a higher precedence than the concatenation operation (,). Enclose concatenation operations in an expression within parentheses to ensure the expression is evaluated correctly. For example, in the following expression, a = b is evaluated before b and c are concatenated:

```
a = b, c;
```

To ensure b and c are concatenated before the result is assigned to c, enclose (b, c) within parentheses as follows:

```
a = (b, c);
```

Using Enumerated Data Types

SystemC Compiler supports enumerated (enum) data types and interprets an enum data type the same way a C++ compiler interprets it. Example 3-2 shows an enum definition.

Example 3-2 Enumerated Data Type

```
enum command_t{
    NONE,
    RED,
    GREEN,
    YELLOW
};
```

Using Aggregate Data Types

To group data types into a convenient aggregate type, define them as a struct type similar to Example 3-3. You need to use all synthesizable data types in a struct in order for the struct to be synthesizable. SystemC Compiler splits the struct type into individual elements for synthesis.

Example 3-3 Aggregate Data Type

```
struct package {
    sc_uint<8> command;
    sc_uint<8> address;
    sc_uint<12> data;
}
```

Using C++ Data Types

The native C++ data types, such as bool, char, int, long, short, unsigned char, unsigned int, unsigned long, and unsigned short, have a platform-specific size. SystemC Compiler synthesizes these types to have the width dictated by your platform.

In some situations, SystemC Compiler can determine that fewer bits are required in hardware than is specified by the data type, which produces a higher-quality result after synthesis. For example, if a unique integer variable is declared as a for loop counter, SystemC Compiler can determine the number of bits and build only the required hardware. Example 3-4 shows a unique loop counter variable in bold. SystemC Compiler can determine that three bits are required, and it builds a 3-bit incrementer for variable i.

Example 3-4 Implicit Bit Size Restriction

```
for (int i=0; i < 7; i++){
      ... //loop code
}
```

If a variable is declared outside of the for loop, SystemC Compiler cannot determine the bit size, because the intended use of the variable is not known at the time of declaration. In this situation, SystemC Compiler builds hardware for the platform-specific bit size. Example 3-5 shows code (in bold) where the loop counter variable is declared outside the loop. In such a situation, SystemC Compiler infers that a variable of 32 or 64 bits is required, depending on the platform. Therefore, it is strongly recommended that you use the coding style shown in Example 3-4 instead of the code in Example 3-5.

Example 3-5 Unknown Variable Bit Size

```
int i;
...
for (i=0; i < 7; i++){
        ... //loop code
}
```

Data Members of a Module

Do not use data members for interprocess communication, because it can lead to nondeterminism (order dependencies) during simulation and it can cause mismatches between the results of presynthesis and postsynthesis simulation. Instead of a data member for interprocess communication, use an sc_signal for this purpose.

Example 3-6 shows (in bold) a data member variable named count that is incorrectly used to communicate between the do_count( ) and outregs( ) processes. A value is written to the count variable in the do_count( ) process, and a value is read from the same variable in the outregs( ) process. The order in which the two processes execute cannot be predicted—therefore, you cannot determine whether writing to the count variable is happening before or after count is incremented.

Example 3-6 Incorrect Use of a Data Member Variable for Interprocess Communication

```
/****mem_var_bad.h****/
#include "systemc.h"

SC_MODULE(counter) {
  sc_in<bool> clk;
  sc_in<bool> reset_z;
  sc_out<sc_uint<4> > count_out;
  sc_uint<4> count;              // Member Variable
  SC_CTOR(counter) {
```

```
    SC_METHOD(do_count);
    sensitive_pos << clk;
    sensitive_neg << reset_z;

    SC_METHOD(outregs);
    sensitive_pos << clk;
    sensitive_neg << reset_z;
  }

  void do_count() {
    if (reset.read() == 0)
      count = 0;
    else
      count++;
  }

  void outregs() {
    if (reset.read() == 0)
      count_out.write(0);
    else
      count_out.write(count);
  }

};
```

To eliminate the nondeterminism of count in Example 3-6, change count to an sc_signal, as shown in bold in Example 3-7. Notice that the only change in the code is the type declaration of count.

Example 3-7 Correct Use of a Signal for Interprocess Communication

```
/****mem_var_good.h****/
#include "systemc.h"

SC_MODULE(counter) {
  sc_in<bool> clk;
  sc_in<bool> reset_z;
  sc_out<sc_uint<4> > count_out;

  // Signal for interprocess communication
  sc_signal<sc_uint<4> > count;
  SC_CTOR(counter) {
```

```
    SC_METHOD(do_count);
    sensitive_pos << clk;
    sensitive_neg << reset_z;

    SC_METHOD(outregs);
    sensitive_pos << clk;
    sensitive_neg << reset_z;
  }

  void do_count() {
    if (reset.read() == 0)
      count = 0;
    else
      count++;
  }

  void outregs() {
    if (reset.read() == 0)
      count_out.write(0);
    else
      count_out.write(count);
  }

};
```

Recommendations About Refinement

The following practices are recommended during refinement:

- After each step in refinement, reverify your design to ensure that you did not introduce errors during that step.
- Although it is recommended that you thoroughly refine at each refinement stage, you might prefer a different refinement technique. For example, during data refinement, you can refine one data type at a time and evaluate the impact on synthesizability and the quality of results with SystemC Compiler. Similarly, you might want to replace one nonsynthesizable construct with a synthesizable construct and reverify the design before replacing the next nonsynthesizable construct.

4

RTL Coding Guidelines

This chapter provides SystemC RTL coding guidelines. The examples in this chapter use the lsi_10k sample target library provided in the $SYNOPSYS/libraries/syn directory.

It contains the following sections:

- Register Inference
- Multibit Inference
- Multiplexer Inference
- Three-State Inference
- Loops
- State Machines

Register Inference

Register inference allows you to use sequential logic in your designs and keep your designs technology-independent. A register is an array of 1-bit memory devices. A latch is a level-sensitive memory device, and a flip-flop is an edge-triggered memory device. Use the coding guidelines in this section to control flip-flop and latch inference.

As a recommended design practice, whenever you infer registers, ensure that the clock and data inputs to the registers can be directly controlled from the ports of the design. This ensures that you can initialize your design easily during simulation as well as in the actual circuit. You can, of course, infer registers with set and reset, which makes the task of register initialization easier and is highly recommended.

Flip-Flop Inference

SystemC Compiler can infer D flip-flops, JK flip-flops, and toggle flip-flops. The following sections provide details about each of these flip-flop types.

Simple D Flip-Flop

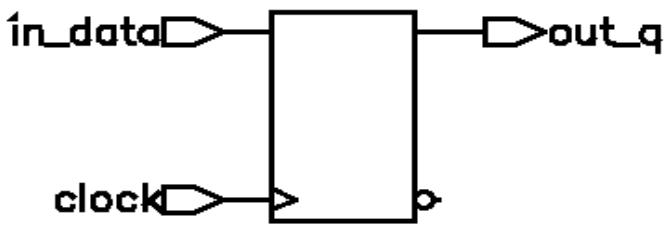
To infer a simple D flip-flop, make the SC_METHOD process sensitive to only one edge of the clock signal. To infer a positive-edge-triggered flip-flop, make the process sensitive to the positive edge of the clock, and make the process sensitive to the negative edge to infer a negative-edge-triggered flip-flop.

SystemC Compiler creates flip-flops for all the variables that are assigned values in the process. Example 4-1 is a common SC_METHOD process description that infers a flip-flop. Figure 4-1 shows the inferred flip-flop.

Example 4-1 Inferring a Positive-Edge-Triggered Flip-Flop

```
/* Positive-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff1) {
  sc_in<bool> in_data;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  // Method for D-flip-flop
  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff1) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock;
  }
};

void dff1::do_dff_pos()
{
  out_q.write(in_data.read());
}
```

Figure 4-1 Inferred Positive-Edge-Triggered Flip-Flop



D Flip-Flop With an Active-High Asynchronous Set or Reset

To infer a D flip-flop with an asynchronous set or reset, include edge expressions for the clock and the asynchronous signals in the sensitivity list of the SC_METHOD process constructor. Specify the asynchronous signal conditions with an if statement in the SC_METHOD process definition. Example 4-2 shows a typical asynchronous specification. Specify the asynchronous branch conditions before you specify the synchronous branch conditions.

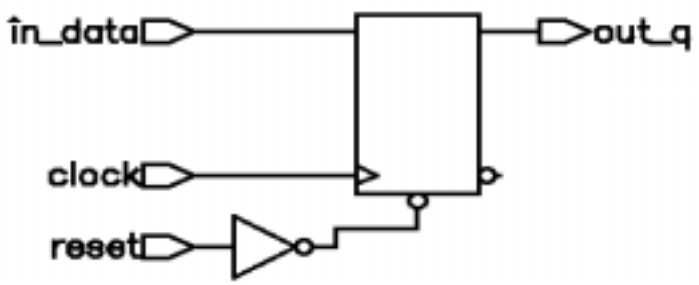
Example 4-2 is the SystemC description for a D flip-flop with an active-high asynchronous reset. Figure 4-2 shows the inferred flip-flop.

Example 4-2 D Flip-Flop With an Active-High Asynchronous Reset

```
/* Positive-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff3) {
  sc_in<bool> in_data, reset;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff3) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock << reset;
  }
};

void dff3::do_dff_pos () {
   if (reset.read())
     out_q.write(0);
```

```
    else
      out_q.write(in_data.read());
  }
```

Figure 4-2 D Flip-Flop With an Active-High Asynchronous Reset



D Flip-Flop With an Active-Low Asynchronous Set or Reset

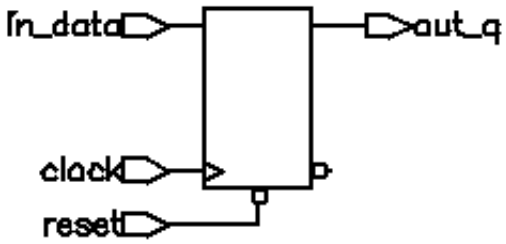
Example 4-3 is a SystemC description for a D flip-flop with an active-low asynchronous reset. Figure 4-3 shows the inferred flip-flop.

Example 4-3 D Flip-Flop With an Active-Low Asynchronous Reset

```
/* Positive-edge-triggered DFF
   with active-low reset */

#include "systemc.h"

SC_MODULE (dff3a) {
  sc_in<bool> in_data, reset;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  void do_dff_pos ();

  // Constructor
```

```
  SC_CTOR (dff3a) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock;
    sensitive_neg << reset;
  }
};

void dff3a::do_dff_pos () {
   if (reset.read() == 0)
     out_q.write(0);
   else
     out_q.write(in_data.read());
  }
```

Figure 4-3 D Flip-Flop With an Active-Low Asynchronous Reset



D Flip-Flop With Active-High Asynchronous Set and Reset

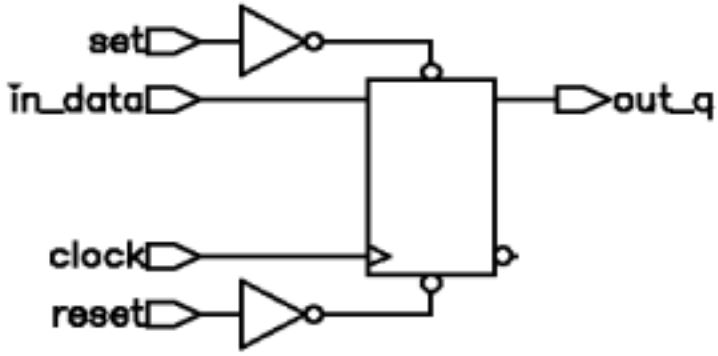
Example 4-4 is a SystemC description for a D flip-flop with active-high asynchronous set and reset ports. Figure 4-4 shows the inferred flip-flop.

An implied priority exists between set and reset, and reset has priority. This priority is not guaranteed, because it can be implemented differently in various technology libraries. To ensure the correct behavior, assign a high value to either the set or reset at one time, but not to both at the same time.

Example 4-4 Flip-Flop With Asynchronous Set and Reset

```
/* Positive-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff4) {
  sc_in<bool> in_data, reset, set;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  void do_dff_pos ();

  // Constructor
  SC_CTOR (dff4) {
    SC_METHOD (do_dff_pos);
    sensitive_pos << clock << reset << set;
  }
};

void dff4::do_dff_pos () {
   if (reset.read())
     out_q.write(0);
   else if (set.read())
     out_q.write(1);
   else
     out_q.write(in_data.read());
  }
```

Figure 4-4 Flip-Flop With Asynchronous Set and Reset



D Flip-Flop With Synchronous Set or Reset

The previous examples illustrated how to infer a D flip-flop with asynchronous controls—one way to initialize or control the state of a sequential device. You can also synchronously reset or set a flip-flop.

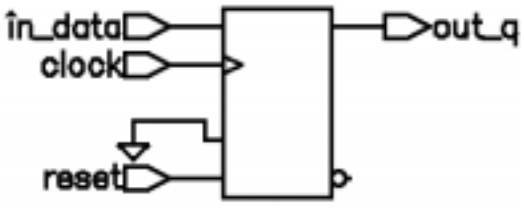
If the target technology library does not have a D flip-flop with a synchronous reset, a D flip-flop with synchronous reset logic as the input to the D pin of the flip-flop is inferred. If the reset (or set) logic is not directly in front of the D pin of the flip-flop, initialization problems can occur during gate-level simulation of the design.

To specify a synchronous set or reset input, do not include it in the sensitivity list. Describe the synchronous set or reset test and action in an if statement. Example 4-5 is a SystemC description for a D flip-flop with synchronous reset. Figure 4-5 shows the inferred flip-flop.

Example 4-5 D Flip-Flop With Synchronous Reset

```
/* Positive-edge-triggered DFF */

#include "systemc.h"

SC_MODULE (dff5) {
  sc_in<bool> in_data, reset;
  sc_out<bool> out_q;
  sc_in<bool> clock;    // clock port

  // Method for D-flip-flop
  void dff ();

  // Constructor
  SC_CTOR (dff5) {
    SC_METHOD (dff);
    sensitive_pos << clock;
  }
};

void dff5::dff()
{
  if (reset.read())
       out_q.write(0);
  else
       out_q.write(in_data.read());
}
```

Figure 4-5 D Flip-Flop With Synchronous Reset



Inferring JK Flip-Flops

Use a switch…case statement to infer JK flip-flops.

JK Flip-Flop With Synchronous Set and Reset. Example 4-6 is the SystemC code that implements the JK flip-flop truth table described in Table 4-1. In the JK flip-flop, the J and K signals are similar to active-high synchronous set and reset. Figure 4-6 shows the inferred flip-flop.

Table 4-1 Positive-Edge-Triggered JK Flip-Flop Truth Table

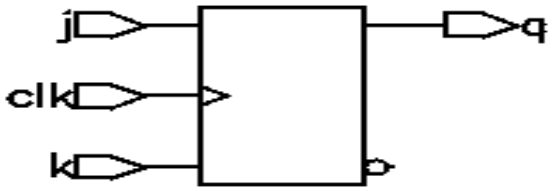
J	K	CLK	Qn+1
0	0	Rising	Qn
0	1	Rising	0
1	0	Rising	1
1	1	Rising	$\overline{Qn}$
X	X	Falling	Qn

Example 4-6 JK Flip-Flop

```
/* Positive-edge-triggered JK FF */

#include "systemc.h"

SC_MODULE (jkff1) {
  sc_in<bool> j, k;
  sc_inout<bool> q;  // inout to read q for toggle
  sc_in<bool> clk;   // clock port

  // Method for D-flip-flop
  void jk_flop ();

 // Constructor
  SC_CTOR (jkff1) {
```

```
    SC_METHOD (jk_flop);
    sensitive_pos << clk;
  }
};
void jkff1::jk_flop() {
    sc_uint<2> temp;          //temp to create vector
    temp[1] = j.read( );
    temp[0] = k.read( );
    switch(temp) {
    case 0x1: q.write(0);      // write a zero
      break;
    case 0x2: q.write(1);      // write a 1
      break;
    case 0x3:                  // toggle
      q.write(!q.read());
      break;
    default: break;           // no change
    }
}
```

Figure 4-6 JK Flip-Flop

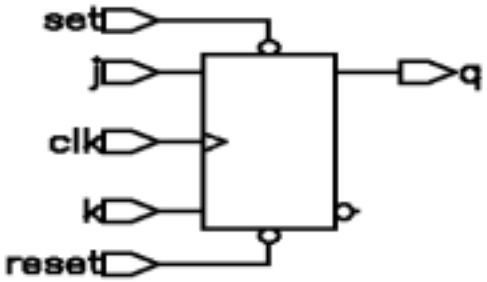


JK Flip-Flop With Asynchronous Set and Reset. Example 4-7 is a SystemC description for a JK flip-flop with an active-low asynchronous set and reset. To specify an asynchronous set or reset, specify the signal in the sensitivity list as shown in Example 4-7. Figure 4-7 shows the inferred flip-flop.

Example 4-7 JK Flip-Flop With Asynchronous Set and Reset

```
/* Positive-edge-triggered JKFF */

#include "systemc.h"

SC_MODULE (jkff2) {
  sc_in<bool> j, k, set, reset;
  sc_inout<bool> q;   // inout to read q for toggle
  sc_in<bool> clk;    // clock port

  // Method for D-flip-flop
  void jk_flop ();

  // Constructor
  SC_CTOR (jkff2) {
    SC_METHOD (jk_flop);
    sensitive_pos << clk;
    sensitive_neg << set << reset;
  }
};
void jkff2::jk_flop() {
      sc_uint<2> temp;  //temp to create vector
      if (reset.read()==0)
           q.write(0);  // reset
      else if (set.read()==0)
           q.write(1);  // set
      else {
           temp[1] = j.read();
           temp[0] = k.read();
           switch(temp) {
             case 0x1: q.write(0);  // write zero
               break;
             case 0x2: q.write(1);  // write a 1
               break;
             case 0x3:              // toggle
               q.write(!q.read());
               break;
             default: break;        // no change
           }
     }
}
```

Figure 4-7 JK Flip-Flop With Asynchronous Set and Reset



Inferring Toggle Flip-Flops

This section describes the toggle flip-flop with an asynchronous set and the toggle flip-flop with an asynchronous reset.

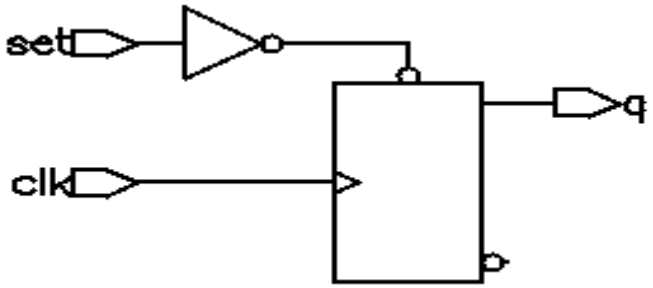
Toggle Flip-Flop With Asynchronous Set. Example 4-8 is a description for a toggle flip-flop with asynchronous set. The asynchronous set signal is specified in the sensitivity list. Figure 4-8 shows the flip-flop.

Example 4-8 Toggle Flip-Flop With Asynchronous Set

```
#include "systemc.h"

SC_MODULE( tff1 ) {
  sc_in<bool> set, clk;
  sc_inout<bool> q;    // inout to read q for toggle

  void t_async_set_fcn ();

 SC_CTOR( tff1 ) {
    SC_METHOD( t_async_set_fcn);
    sensitive_pos << clk << set;
  }
};
```

```
void tff1::t_async_set_fcn () {
      if (set.read())
          q.write(1);
      else
          q.write(!q.read());
}
```

Figure 4-8 Toggle Flip-Flop With Asynchronous Set



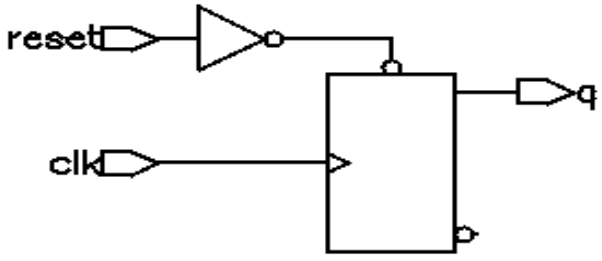
Toggle Flip-Flop With Asynchronous Reset. Example 4-9 is a SystemC description for a toggle flip-flop with asynchronous reset. The asynchronous reset signal is specified in the sensitivity list. Figure 4-9 shows the inferred flip-flop.

Example 4-9 Toggle Flip-Flop With Asynchronous Reset

```
#include "systemc.h"

SC_MODULE( tff2 ) {
  sc_in<bool> reset, clk;
  sc_inout<bool> q;  // to read q for toggle

  void t_async_reset_fcn();

 SC_CTOR( tff2 ) {
    SC_METHOD( t_async_reset_fcn);
    sensitive_pos << clk << reset;
  }
```

```
};

void tff2::t_async_reset_fcn () {
      if (reset.read())
          q.write(0);
      else
          q.write(!q.read());
}
```

Figure 4-9 Toggle Flip-Flop With Asynchronous Reset



Latch Inference

In simulation, a signal or variable holds its value until that value is reassigned. A latch implements the ability to hold a state in hardware. SystemC Compiler supports inference of set/reset (SR) and delay (D) latches.

You can unintentionally infer latches from your SystemC code, which can add unnecessary hardware. SystemC Compiler infers a D latch when your description has an incomplete assignment in an if…else or switch…case statement. To avoid creating a latch, specify all conditions in if…else and switch…case statements, and assign all variables in each branch.

Inferring a D Latch From an If Statement

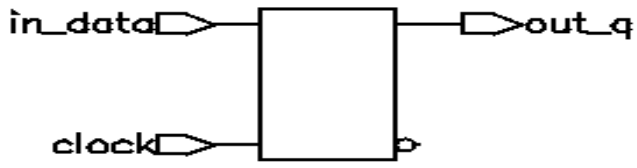
An if statement infers a D latch when there is no else clause, as shown in Example 4-10. The SystemC code specifies a value for output out_q only when the clock has a logic 1 value, and it does not specify a value when the clock has a logic 0 value. As a result, output out_q becomes a latched value. Figure 4-10 shows the schematic of the inferred latch.

Example 4-10 D Latch Inference Using an if Statement

```
#include "systemc.h"

SC_MODULE( d_latch1 ) {
  sc_in<bool> in_data;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  // Method process
  void d_latch_fcn () {
    if (clock.read())
       out_q.write(in_data.read());
  }
```

```
  // Constructor
  SC_CTOR( d_latch1 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock;
  }
};
```

Figure 4-10 D Latch Inferred From an if Statement



Inferring An SR Latch

SR latches are difficult to test, so use them with caution. If you use SR latches, verify that the inputs are hazard-free and do not generate glitches. During synthesis, SystemC Compiler does not ensure that the logic driving the inputs is hazard-free.

Example 4-11 is the SystemC code that implements the truth table in Table 4-2 for an SR latch. Figure 4-11 shows the inferred SR latch.

Output y is unstable when both inputs are at a logic 0 value, so you need to include a check in the SystemC code to detect this condition during simulation. SystemC Compiler does not support these checks.

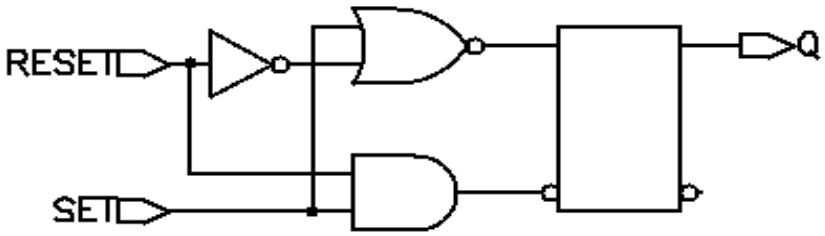
Table 4-2 Truth Table for the SR Latch (NAND Type)

set	reset	Q
0	0	Not stable
0	1	1
1	0	0
1	1	Q

Example 4-11 SR Latch

```
/* SR_LATCH-latch */

#include "systemc.h"

SC_MODULE( sr_latch ) {
  sc_in<bool> RESET, SET;
  sc_out<bool> Q;

  void sr_latch_fcn () {
    if (RESET.read() == 0)
      Q.write(0);
    else if (SET.read() == 0)
      Q.write(1);
  }

  SC_CTOR( sr_latch ) {
    SC_METHOD( sr_latch_fcn);
    sensitive << RESET << SET;
  }
};
```

Figure 4-11 SR Latch



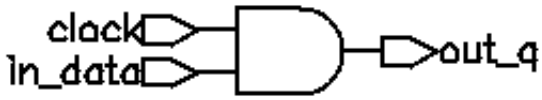
Avoiding Latch Inference

To avoid latch inference, assign a value to a signal for all cases in a conditional statement. Example 4-13 shows addition of an else clause to avoid the latch inferred by the if statement in Example 4-10, and Figure 4-12 shows the resulting schematic.

Example 4-12 Adding an Else Clause to Avoid Latch Inference

```
#include "systemc.h"

SC_MODULE( d_latch1a ) {
  sc_in<bool> in_data;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  // Method process
  void d_latch_fcn () {
    if (clock.read())
       out_q.write(in_data.read());
    else
       out_q.write(false);
  }

  // Constructor
  SC_CTOR( d_latch1a ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock;
  }
};
```

Figure 4-12 Avoiding Latch Inference by Adding Else Clause

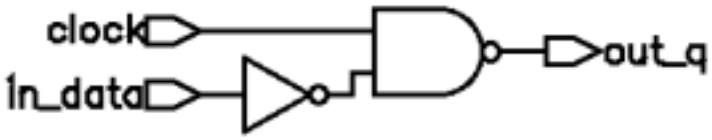


You can also avoid latch inference by assigning a default value to the output port. Example 4-13 shows the setting of a default value to avoid the latch inferred by the if statement in Example 4-10, and Figure 4-13 shows the resulting schematic.

Example 4-13 Setting a Default Value to Avoid Latch Inference

```
#include "systemc.h"

SC_MODULE( d_latch1 ) {
  sc_in<bool> in_data;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  // Method process
  void d_latch_fcn () {
    out_q.write(1);     // set a default
    if (clock.read())
       out_q.write(in_data.read());
  }

  // Constructor
  SC_CTOR( d_latch1 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock;
  }
};
```

Figure 4-13 Avoiding Latch Inference by a Default Value



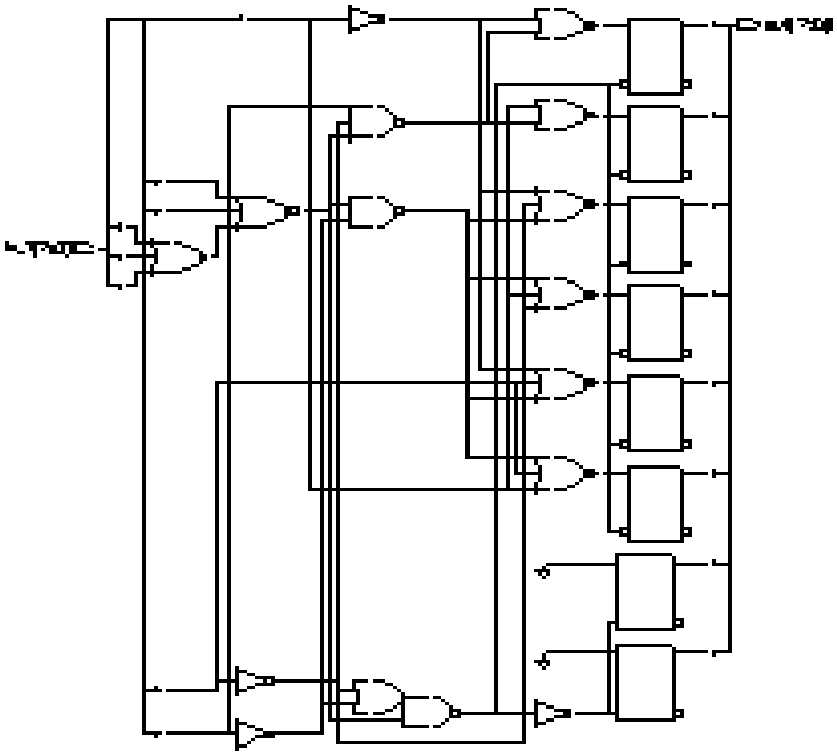
Inferring a Latch From a Switch Statement

Example 4-14 shows a switch statement that infers D latches because the switch statement does not provide assignments to the out port for all possible values of the in_i input. Figure 4-14 shows the inferred latches.

Example 4-14 Latch Inference From a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2 ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    switch (in_i.read()) {
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    }
  }

  // Constructor
  SC_CTOR( d_latch2 ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-14 Latch Inference From a switch Statement



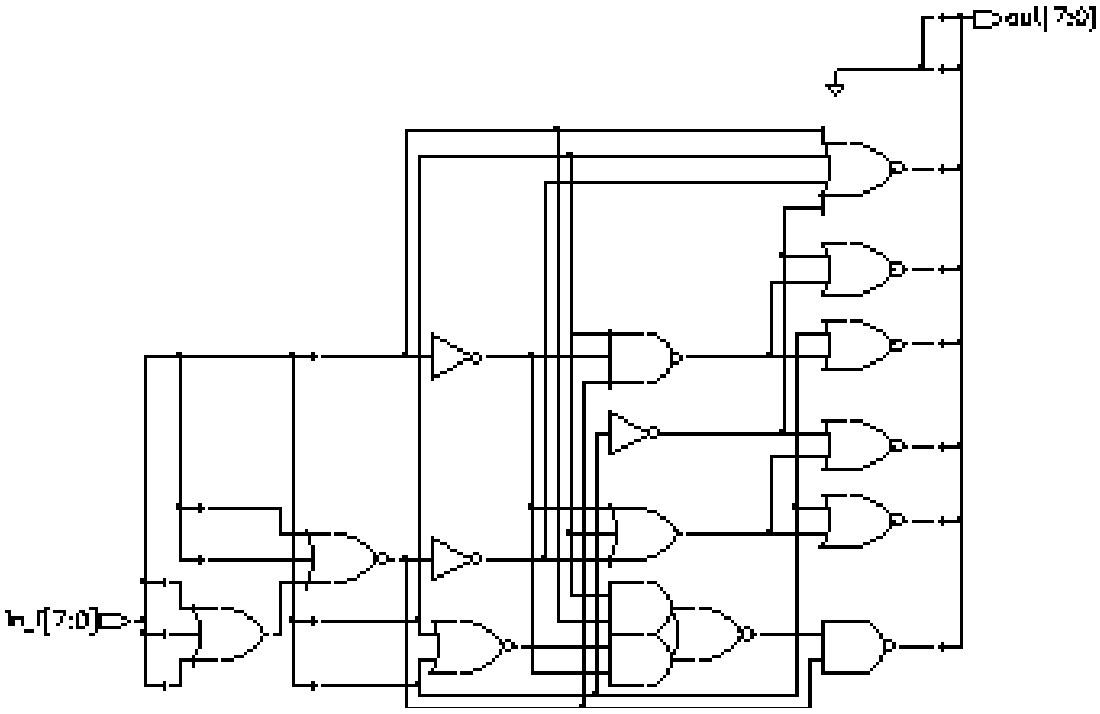
To avoid latch inference caused by the incomplete switch statement in Example 4-14, add a default case statement, as shown in Example 4-15. Figure 4-15 shows the resulting schematic.

Example 4-15 Avoiding Latch Inference From a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2a ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    switch (in_i.read()) {
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
```

```
      case 2: out.write(0x04); break;
      case 3: out.write(0x10); break;
      case 4: out.write(0x20); break;
      case 5: out.write(0x40); break;
      default: out.write(0x01);
      }
    }

    // Constructor
    SC_CTOR( d_latch2a ) {
      SC_METHOD( d_latch_fcn);
      sensitive (in_i);
    }
};
```

Figure 4-15 Avoiding Latch Inference by Adding a Default Case to a switch Statement

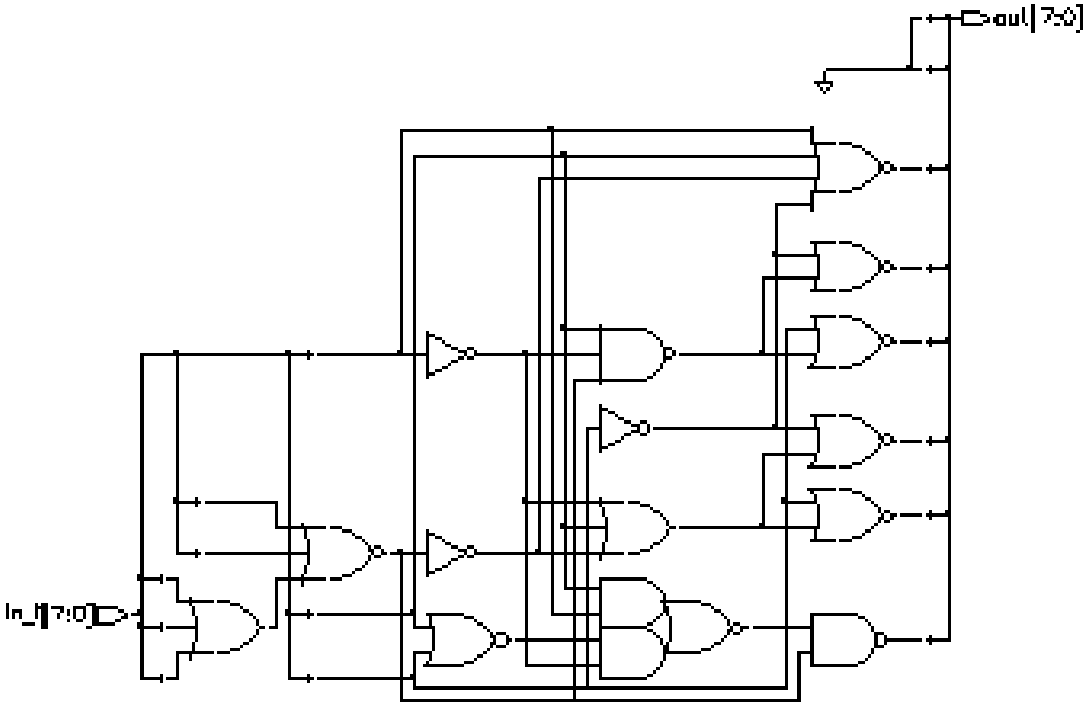


You can also avoid latch inference caused by the incomplete switch statement in Example 4-14 by writing a default value to the output port, as shown in Example 4-16. Figure 4-16 shows the resulting schematic.

Example 4-16 Set a Default Value to Avoid Latch Inference From a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2b ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    out.write(1);  // Set default value
    switch (in_i.read()) {
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    }
  }

  // Constructor
  SC_CTOR( d_latch2b ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-16 Avoiding Latch Inference by Setting a Default Case Before a switch Statement



Priority Encoding

Switch…case and if…else conditional statements are priority-encoded in simulation. Priority-encoded hardware is rarely needed, and it can add unnecessary gates and time to the synthesized design.

SystemC Compiler defaults to priority-encoded logic for a switch statement when

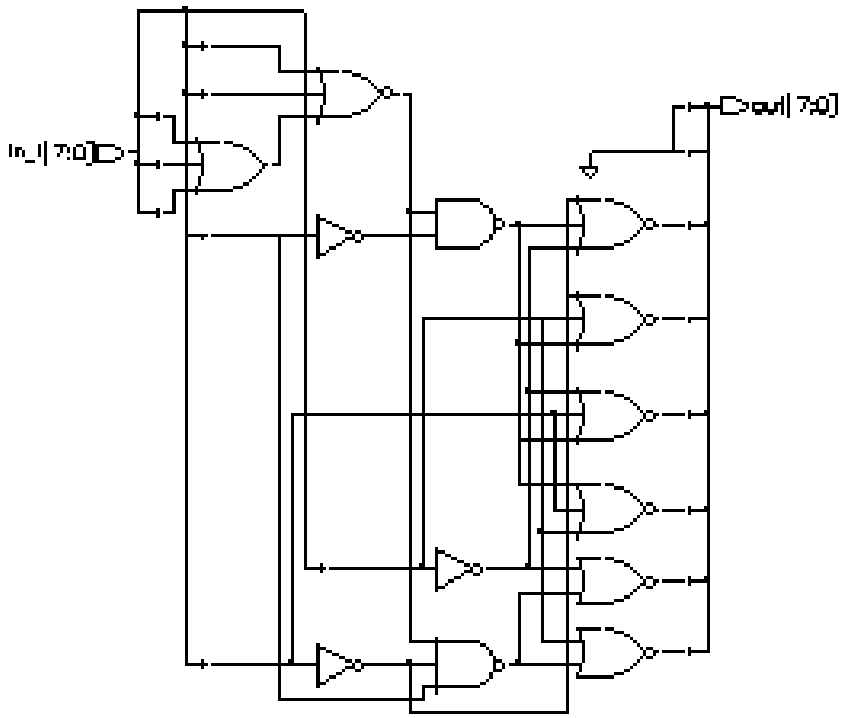
- You do not define all cases in a switch statement
- All cases are not mutually exclusive

In addition to defining a default case (Example 4-15 on page 4-22) and setting a default value (Example 4-16 on page 4-24), you can instruct SystemC Compiler that other cases are not necessary for a switch statement by adding the `full_case` compiler directive in your code. Example 4-17 uses the `full_case` directive, and Figure 4-17 shows the resulting schematic.

Example 4-17 Using the full_case Compiler Directive With a switch Statement

```
#include "systemc.h"

SC_MODULE( d_latch2c ) {
  sc_in<unsigned char> in_i;
  sc_out<unsigned char> out;

  // Method process
  void d_latch_fcn () {
    switch (in_i.read()) {
    // synopsys full_case
    case 0: out.write(0x01); break;
    case 1: out.write(0x02); break;
    case 2: out.write(0x04); break;
    case 3: out.write(0x10); break;
    case 4: out.write(0x20); break;
    case 5: out.write(0x40); break;
    }
  }

  // Constructor
  SC_CTOR( d_latch2c ) {
    SC_METHOD( d_latch_fcn);
    sensitive (in_i);
  }
};
```

Figure 4-17 Using the full_case Compiler Directive With a switch Statement



Active-Low Set and Reset

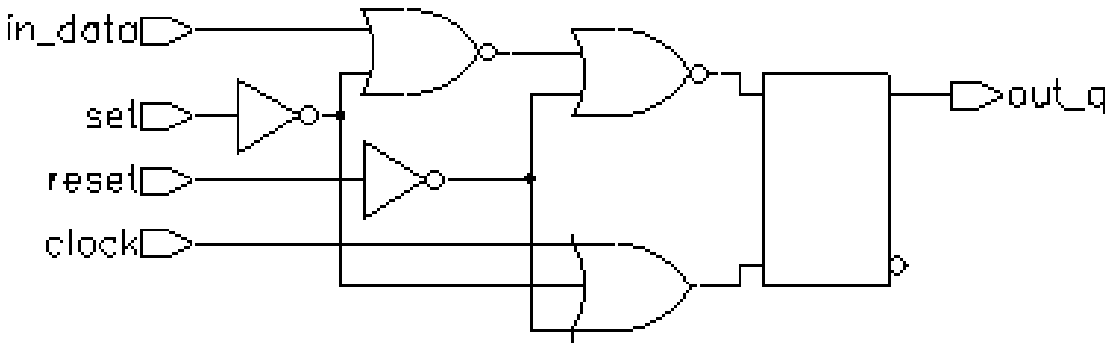
To instruct SystemC Compiler to implement all signals in the group as active-low, add a check to the SystemC code to ensure that the group of signals has only one active-low signal at a given time. SystemC Compiler does not produce any logic to check this assertion.

Example 4-18 shows a latch with an active-low set and reset. Figure 4-18 shows the resulting schematic.

Example 4-18 Latch With Active-Low Set and Reset

```
#include "systemc.h"

SC_MODULE( d_latch6a ) {
  sc_in<bool> in_data, set, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn (){
    infer_latch: {
      if (reset.read() == 0)
        out_q.write(0);
      else if (set.read() == 0)
        out_q.write(1);
      else if (clock.read())
        out_q.write(in_data.read());
     }
  }
  // Constructor
  SC_CTOR( d_latch6a ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set << reset;
  }
};
```

Figure 4-18 Latch With Active-Low Set and Reset



Active-High Set and Reset

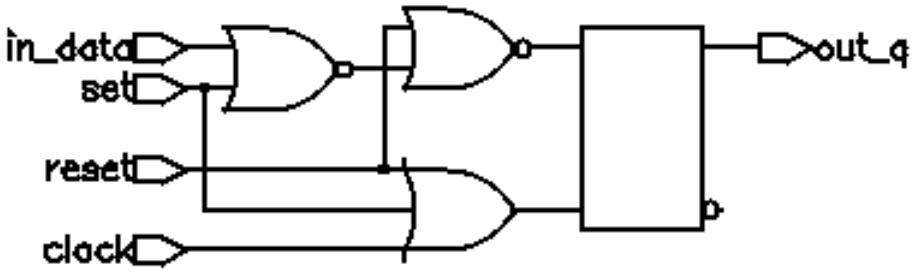
To instruct SystemC Compiler to implement all signals in the group as active-high, add a check to the SystemC code to ensure that the group of signals has only one active-high signal at a given time. SystemC Compiler does not produce any logic to check this assertion.

Example 4-18 shows a latch with the set and reset specified as active-high. Figure 4-18 shows the resulting schematic.

Example 4-19 Latch With Active-High Set and Reset

```
#include "systemc.h"

SC_MODULE( d_latch7a ) {
  sc_in<bool> in_data, set, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn (){
    infer_latch: {
      if (reset.read())
        out_q.write(0);
      else if (set.read())
        out_q.write(1);
      else if (clock.read())
        out_q.write(in_data.read());
     }
  }
  // Constructor
  SC_CTOR( d_latch7a ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set << reset;
  }
};
```

Figure 4-19 Latch With Active-High Set and Reset



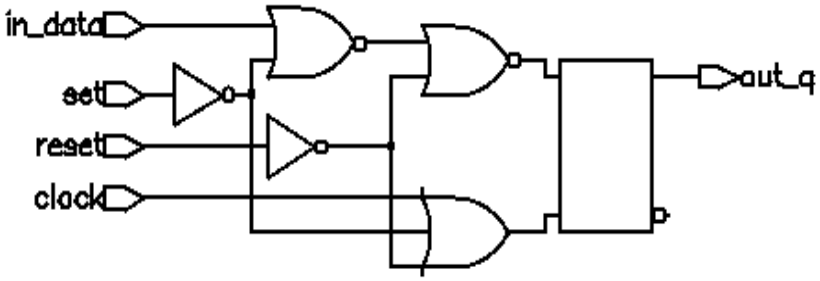
D Latch With an Asynchronous Set and Reset

Example 4-20 is a SystemC description for a D latch with an active-low asynchronous set and reset. Figure 4-20 shows the inferred latch.

Example 4-20 Latch With Asynchronous Set and Reset

```
#include "systemc.h"

SC_MODULE( d_latch6 ) {
  sc_in<bool> in_data, set, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn (){
      if (reset.read() == 0)
        out_q.write(0);
      else if (set.read() == 0)
        out_q.write(1);
      else if (clock.read())
        out_q.write(in_data.read());
  }
  // Constructor
  SC_CTOR( d_latch6 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set << reset;
  }
};
```

Figure 4-20 Latch With Asynchronous Set and Reset



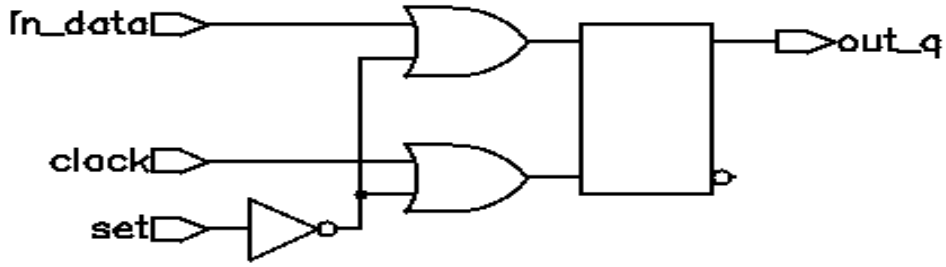
D Latch With an Asynchronous Set

Example 4-21 is a SystemC description for a D latch with an asynchronous set. Figure 4-21 shows the inferred latch.

Example 4-21 Latch With Asynchronous Set

```
#include "systemc.h"

SC_MODULE( d_latch4 ) {
  sc_in<bool> in_data, set;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn () {
    if (set.read() == 0)
      out_q.write( 1 );
    else if (clock.read())
      out_q.write(in_data.read());
  }

  // Constructor
  SC_CTOR( d_latch4 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << set;
  }
};
```

Figure 4-21 Latch With Asynchronous Set



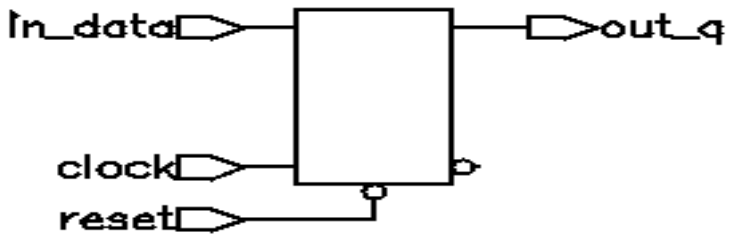
D Latch With an Asynchronous Reset

Example 4-22 is a SystemC description for a D latch with an asynchronous reset. Figure 4-22 shows the inferred latch.

Example 4-22 Latch With Asynchronous Reset

```
#include "systemc.h"

SC_MODULE( d_latch5 ) {
  sc_in<bool> in_data, reset;
  sc_in<bool> clock;
  sc_out<bool> out_q;

  void d_latch_fcn () {
    if (reset.read() == 0)
      out_q.write(0);
    else if (clock.read())
      out_q.write(in_data.read());
  }
  // Constructor
  SC_CTOR( d_latch5 ) {
    SC_METHOD( d_latch_fcn);
    sensitive << in_data << clock << reset;
  }
};
```

Figure 4-22 Latch With Asynchronous Reset



Understanding the Limitations of Register Inference

SystemC Compiler cannot infer the following components:

- Flip-flops and latches with three-state outputs
- Flip-flops with bidirectional pins
- Flip-flips with multiple clock inputs
- Multiport latches

You can instantiate these components in your SystemC description. SystemC Compiler interprets these flip-flops and latches as black boxes.

To instantiate one of these components in your SystemC RTL netlist, you need to create a dummy SystemC module with the same module name and port names as the cell in the technology library that you want to instantiate. The module and port names are case-sensitive and must exactly match the cell names. You do not need to describe the module's function, because Design Compiler replaces it with the actual library cell.

Example 4-23 shows a dummy module for the or2c1 cell from the tc6a_cbacore sample technology library. An instance of the or2c1 module named my_gate is created in the gate module.

Example 4-23 Instantiating a Register That Cannot Be Inferred

```
/****gate.h****/


/*****************************************
 *  This example shows how to instantiate
 *  an or2c1 gate from the tc6a_cbacore
 *  library in a SystemC RTL netlist.
 *****************************************/
#include "systemc.h"

SC_MODULE(or2c1) {
  sc_in<bool>  A, B;
  sc_out<bool> Y;

  SC_CTOR(or2c1) {}
};

SC_MODULE(gate) {
  sc_in<bool>          clk, reset;
  sc_in<sc_uint<8> >   data;
  sc_out<bool>         match;

  sc_signal<bool> a_match, b_match;

  /*
   *   RTL processes
   */
  void match_a();
  void match_b();

  /*
   *   Pointer for block allocation
   */
  or2c1 *my_gate;
```

```
  SC_CTOR(gate) {
    /*
     *   Instantiate and hook up the gate
     */
    my_gate = new or2c1("my_gate");
    my_gate->A(a_match);
    my_gate->B(b_match);
    my_gate->Y(match);

    SC_METHOD(match_a);
    sensitive_pos << clk;
    sensitive_neg << reset;

    SC_METHOD(match_b);
    sensitive_pos << clk;
    sensitive_neg << reset;
  }
};

/****gate.cc****/
#include "gate.h"

void gate::match_a() {
  if (reset.read() == 0) {
    a_match = 0;
  } else {
    if (data.read() == 3) {
      a_match = 1;
    } else {
      a_match = 0;
    }
  }
}

void gate::match_b() {
  if (reset.read() == 0) {
    b_match = 0;
  } else {
    if (data.read() == 7) {
      b_match = 1;
    } else {
      b_match = 0;
```

```
    }
  }
}
```

To perform RTL synthesis and instantiate the or2c1 cell,

1. Elaborate the gate module that contains the dummy or2c1 module.

   ```
   dc_shell>  compile_systemc -rtl -rtl_format db gate.cc
   ```

2. Remove the dummy module.

   ```
   dc_shell> remove_design or2c1
   ```

3. Set the current design to be the gate module.

   ```
   dc_shell> current_design gate
   ```

4. Remove the current links and create new links for the design. This links the or2c1 cell from the technology library.

   ```
   dc_shell> link
   ```

5. Instruct Design Compiler not to touch the my_gate instantiation of the or2c1 cell in the design.

   ```
   dc_shell> set_dont_touch find(cell, my_gate)
   ```

6. Compile the design to gates.

   ```
   dc_shell> compile
   ```

Multibit Inference

A multibit component (MBC), such as a 16-bit register, reduces the area and power in a design. The primary benefits of MBCs are to create a more uniform structure for layout during place and route.

Multibit inference allows you to map registers, multiplexers, and three-state cells to regularly structured logic or multibit library cells. Multibit library cells (macro cells, such as 16-bit banked flip-flops) have these advantages:

- Smaller area and delay, due to shared transistors (as in select or set/reset logic) and optimized transistor-level layout. In the use of single-bit components, the select or set/reset logic is repeated in each single-bit component.
- Reduced clock skew in sequential gates, because the clock paths are balanced internally in the hard macro implementing the MBC.
- Lower power consumption by the clock in sequential banked components, due to reduced capacitance driven by the clock net.
- Better performance, due to the optimized layout within the MBC.
- Improved regular layout of the data path.

To direct SystemC Compiler to infer multibit components,

- Add the `infer_multibit` or `dont_infer_ multibit` compiler directive (see “Multibit Inference Compiler Directives” on page A-3) to individual ports or signals in the SystemC description.

- Or define the dc_shell `hdlin_infer_multibit` variable, which specifies that multibit inferencing is allowed for the entire design. The allowed values for `hdlin_infer_multibit` are `default_all`, `default_none`, and `never`. See the `hdlin_infer_multibit` manpage for additional information.

Inferring Multibit

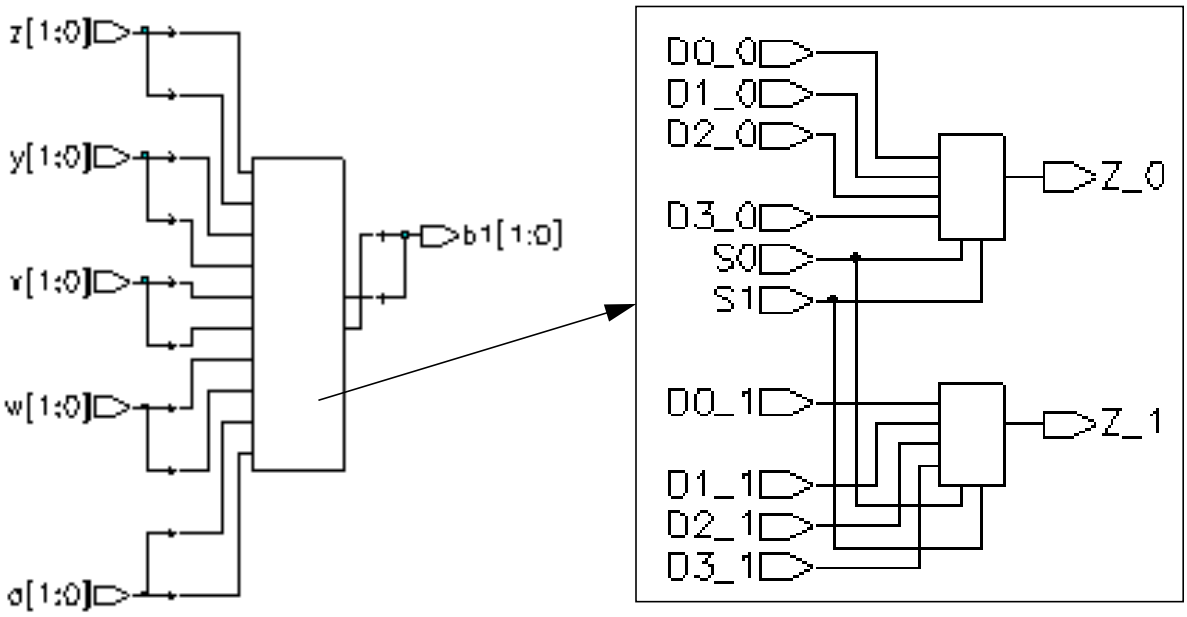
Example 4-24 shows inference of a 2-bit multiplexer, resulting in the schematic in Figure 4-23.

Example 4-24 Inferring a 2-Bit 4-to-1 Multiplexer

```
#include "systemc.h"

SC_MODULE( infer_multibit ) {

  sc_in<sc_uint<2> > a;
  sc_in<sc_int<2> > w;
  sc_in<sc_int<2> > x;
  sc_in<sc_int<2> > y;
  sc_in<sc_int<2> > z;

  sc_out<sc_int<2> > b1; // synopsys infer_multibit "b1"

  void f1 ();

  SC_CTOR( infer_multibit ) {
    SC_METHOD( f1);
    sensitive << a << w << x << y << z;
  }
};

void infer_multibit:: f1 ()
{
  switch (a.read ()) {
  case 3:
    b1.write(w);
    break;
```

```
      case 2:
        b1.write(x);
        break;
      case 1:
        b1.write(y);
        break;
      case 0:
        b1.write(z);
        break;
      }
  }
```

Figure 4-23 Inferring a 2-Bit 4-to-1 Multiplexer



Preventing Multibit Inference

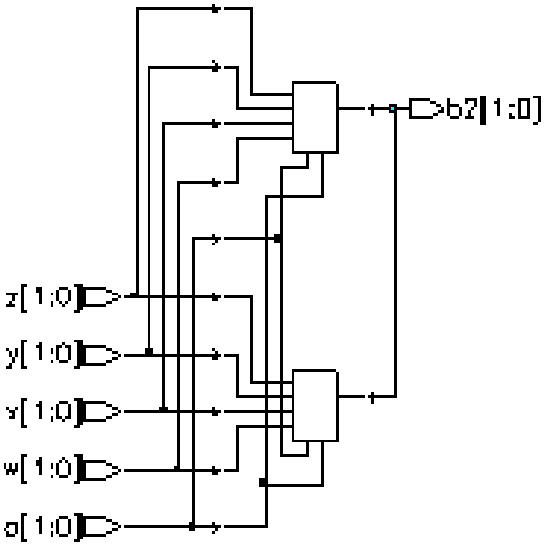
Example 4-25 shows restriction of the description to prevent inference of a 2-bit multiplexer. This restriction results in the schematic in Figure 4-24.

Example 4-25 Do Not Infer a 2-Bit 4-to-1 Multiplexer

```
#include "systemc.h"

SC_MODULE( infer_multibit2 ) {

  sc_in<sc_uint<2> > a;
  sc_in<sc_int<2> > w;
  sc_in<sc_int<2> > x;
  sc_in<sc_int<2> > y;
  sc_in<sc_int<2> > z;

 sc_out<sc_int<2> > b2; // synopsys dont_infer_multibit "b2"

  void f1 ();

  SC_CTOR( infer_multibit2 ) {
    SC_METHOD( f1);
    sensitive << a << w << x << y << z;
  }
};

void infer_multibit2:: f1 ()
{
  switch (a.read ()) {
  case 3:
    b2.write(w);
    break;
  case 2:
    b2.write(x);
    break;
  case 1:
    b2.write(y);
    break;
  case 0:
```

```
      b2.write(z);
      break;
    }
  }
```

Figure 4-24 Do Not Infer a 2-Bit 4-to-1 Multiplexer



Multiplexer Inference

SystemC Compiler can infer a generic multiplexer cell (MUX_OP) from switch statements and if-then-else statements in your SystemC description. SystemC Compiler maps inferred MUX_OPs to multiplexer cells in the target technology library.

The size of the inferred MUX_OP depends on the number of unique cases in the switch statement. If you want to use the multiplexer inference feature, the target technology library must contain at least a 2-to-1 multiplexer.

MUX_OPs are hierarchical cells similar to Synopsys DesignWare components. SystemC Compiler passes the multiplexer inference information to Design Compiler, and Design Compiler determines the MUX_OP implementation during logic synthesis, based on the design constraints. For information about how Design Compiler maps MUX_OPs to multiplexers in the target technology library, see the *Design Compiler Reference Manual: Optimization and Timing Analysis*.

Inferring Multiplexers From a Block of Code

Use the `infer_mux` compiler directive to instruct SystemC Compiler to infer MUX_OPs for all switch statements inside a block of code. In Example 4-26, the `infer_mux` compiler directive is attached to the code block labeled tt, which contains two switch statements. The code block can contain any number of switch statements.

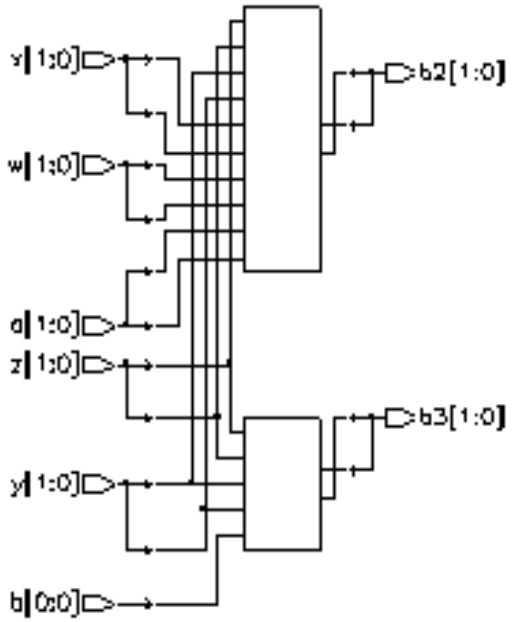
SystemC Compiler infers a MUX_OP for each case in the switch statement. The first switch statement has four unique cases and infers an 4-to-1 MUX_OP. The second switch statement has two unique cases and infers a 2-to-1 MUX_OP. Figure 4-25 shows the inferred multiplexers.

Example 4-26 Multiplexer Inference From a Block of Code

```
#include "systemc.h"

SC_MODULE( infer_mux_blk ) {
  sc_in<sc_uint<2> > a;
  sc_in<sc_uint<1> > b;
  sc_in<sc_int<2> > w, x, y, z;
  sc_out<sc_int<2> > b2, b3;

  void f2 ();

  SC_CTOR( infer_mux_blk ) {

    SC_METHOD( f2);
    sensitive << a <<b << w << x << y << z;
  }
};

//  infer mux for all switch statements in block 'tt'
void infer_mux_blk:: f2 ()
{
  // synopsys infer_mux "tt"
  tt: {
    switch (a.read ()) {
    case 3:
      b2.write(w);
      break;
    case 2:
      b2.write(x);
      break;
    case 1:
      b2.write(y);
      break;
    case 0:
```

```
        b2.write(z);
        break;
      }
      switch (b.read ()) {
      case 1:
        b3.write(y);
        break;
      case 0:
        b3.write(z);
        break;
      }
    }
  }
```

Figure 4-25 Inferring a Multiplexer for a Block



Preventing Multiplexer Inference

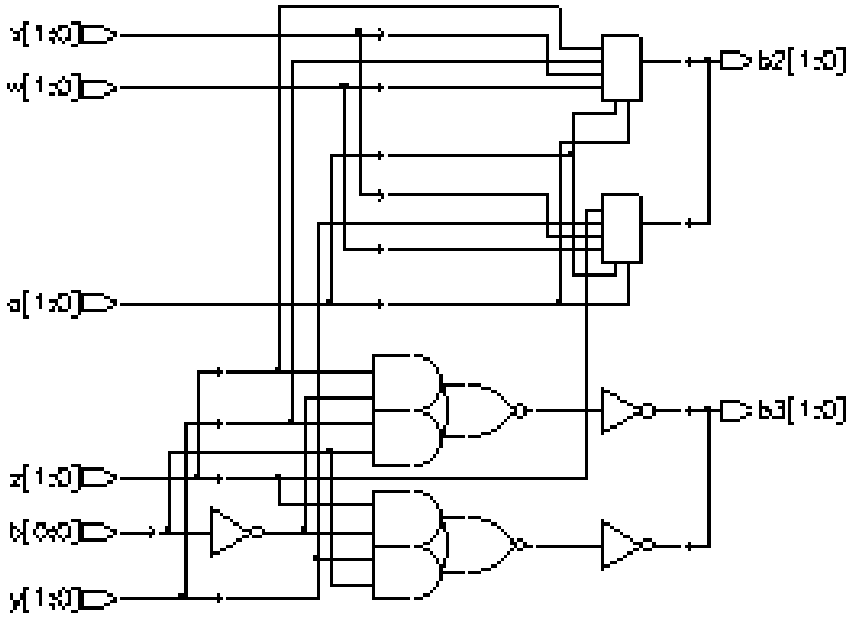
Example 4-27 shows the code from Example 4-26 without the `infer_mux` compiler directive, and Figure 4-26 shows the resulting schematic.

Example 4-27 No Multiplexer Inference From a Block of Code

```
#include "systemc.h"

SC_MODULE( infer_mux_blk ) {
  sc_in<sc_uint<2> > a;
  sc_in<sc_uint<1> > b;
  sc_in<sc_int<2> > w, x, y, z;
  sc_out<sc_int<2> > b2, b3;

  void f2 ();

  SC_CTOR( infer_mux_blk ) {

    SC_METHOD( f2);
    sensitive << a <<b << w << x << y << z;
  }
};

/*  Do not use the infer mux for all switch
    statements in block 'tt' */

void infer_mux_blk:: f2 ()
{
  tt: {
    switch (a.read ()) {
    case 3:
      b2.write(w);
      break;
    case 2:
      b2.write(x);
      break;
    case 1:
      b2.write(y);
      break;
```

```
      case 0:
        b2.write(z);
        break;
      }
      switch (b.read ()) {
      case 1:
        b3.write(y);
        break;
      case 0:
        b3.write(z);
        break;
      }
    }
  }
```

Figure 4-26 Block of Code Without Multiplexer Inference



Inferring a Multiplexer From a Specific Switch Statement

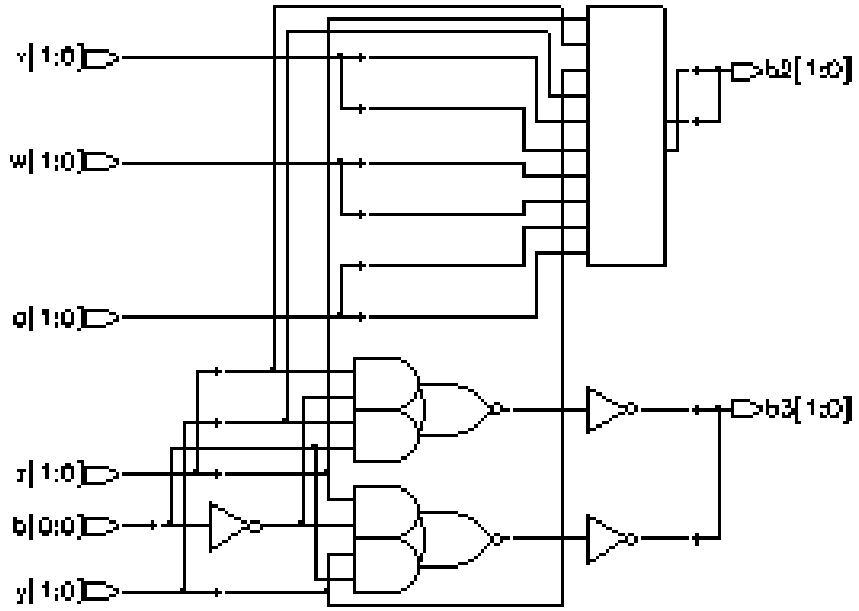
You can also specify the `infer_mux` compiler directive from a single switch statement by placing the compiler directive as the first line inside the switch statement, as shown in Example 4-28. This switch statement reads four unique values, and SystemC Compiler infers a 4-to-1 MUX_OP. Figure 4-27 shows the inferred multiplexer.

Example 4-28 Multiplexer Inference From a Specific switch Statement

```
#include "systemc.h"

SC_MODULE( infer_mux_blk3 ) {
  sc_in<sc_uint<2> > a;
  sc_in<sc_uint<1> > b;
  sc_in<sc_int<2> > w, x, y, z;
  sc_out<sc_int<2> > b2, b3;

  void f2 ();

  SC_CTOR( infer_mux_blk3 ) {

    SC_METHOD( f2);
    sensitive << a <<b << w << x << y << z;
  }
};

/*  Infer mux for only the first switch statement
    in block 'tt' */
void infer_mux_blk3:: f2 ()
{
  tt: {
    switch (a.read ()) { //synopsys infer_mux
    case 3:
      b2.write(w);
      break;
    case 2:
      b2.write(x);
      break;
```

```
      case 1:
        b2.write(y);
        break;
      case 0:
        b2.write(z);
        break;
      }
      switch (b.read ()) {
      case 1:
        b3.write(y);
        break;
      case 0:
        b3.write(z);
        break;
      }
    }
  }
```

Figure 4-27 Inferring a Multiplexer From a Specific switch Statement



Understanding the Limitations of Multiplexer Inference

SystemC Compiler does not infer MUX_OPs for

- if...else statements
- switch statements in while loops

SystemC Compiler infers MUX_OPs for incompletely specified switch statements, but the resulting logic might not be optimal. SystemC Compiler considers the following types of switch statements incompletely specified:

- A switch statement that has a missing case statement branch
- A switch statement that contains an if statement
- A switch statement that contains other switch statements

Three-State Inference

A three-state driver is inferred when you assign the value Z to a variable. The value Z represents the high-impedance state. You can assign high-impedance values to single-bit or bused variables. The assignment must occur in a conditional statement (if or switch) or with the conditional operator (?:). Note that only the sc_logic and sc_lv data types support the value Z.

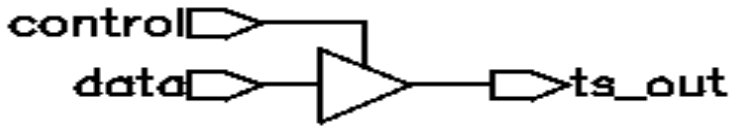
Simple Three-State Inference

Example 4-29 is a SystemC description for a simple three-state driver. Figure 4-28 shows the schematic the code generates.

Example 4-29 Three-State Buffer Inference From a Block of Code

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex1 ) {
  sc_in<bool> control;
  sc_in<sc_logic> data;
  sc_out<sc_logic> ts_out;

  // Method for three-state driver
  void tristate_fcn () {
    if (control.read())
      ts_out.write(data.read());
    else
      ts_out.write('Z');
  }

  // Constructor
  SC_CTOR( tristate_ex1 ) {
    SC_METHOD( tristate_fcn);
    sensitive << control << data;
  }
};
```

Figure 4-28 Schematic of a Simple Three-State Driver

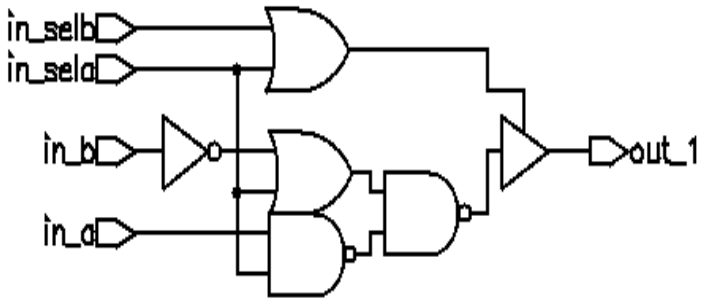


Example 4-30 is a different coding style for three-state inference. In this case, the computed and SystemC Compiler infers a single three-state driver. Figure 4-29 shows the schematic the code generates.

Example 4-30 Inferring One Three-State Driver

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex2 ) {
  sc_in<bool> in_sela, in_selb;
  sc_in<sc_logic> in_a, in_b;
  sc_out<sc_logic> out_1;

  // Method for single three-state driver
  void tristate_fcn () {
    out_1.write('Z');  //default value
    if (in_sela.read())
      out_1.write(in_a.read());
    else if (in_selb.read())
      out_1.write(in_b.read());
  }

  // Constructor
  SC_CTOR( tristate_ex2 ) {
    SC_METHOD( tristate_fcn);
    sensitive << in_sela <<in_selb << in_a << in_b;
  }
};
```

Figure 4-29 Three-State Driver With Gated Data



Registered Three-State Drivers

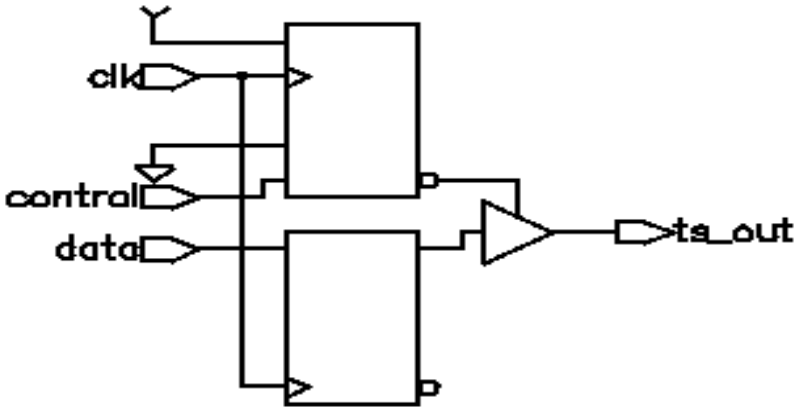
When a variable is registered in the same process in which it is inferred as a three-state, SystemC Compiler also registers the enable pin of the three-state gate. Example 4-31 is an example of this type of code. Figure 4-30 shows the schematic generated by the code.

Example 4-31 Three-State Driver With Registered Enable

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex4 ) {
  sc_in<bool> control;
  sc_in<sc_logic> data;
  sc_out<sc_logic> ts_out;
  sc_in_clk clk;

  // Method for three-state driver
  void tristate_fcn () {
    if (control.read())
      ts_out.write(data.read());
    else
      ts_out.write('Z');
  }

  // Constructor
```

```
  SC_CTOR( tristate_ex4 ) {
    SC_METHOD( tristate_fcn);
    sensitive_pos << clk; // note inferred seq logic
  }
};
```

Figure 4-30 Three-State Driver With Registered Enable



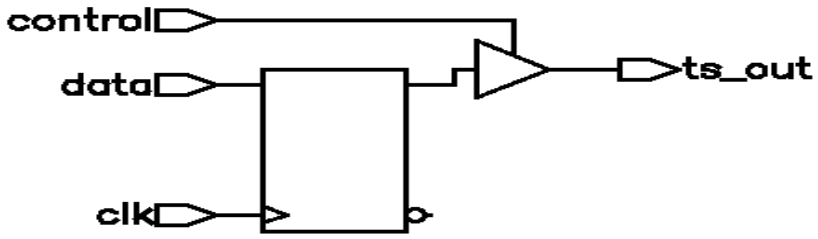
To avoid registering the enable pin, separate the three-state driver inference from the sequential logic inference, using two SC_METHOD processes. Example 4-32 uses two methods to instantiate a three-state gate, with a flip-flop only on the input. Note that the sc_signal temp is used to communicate between the two SC_METHOD processes. Figure 4-31 shows the schematic the code generates.

Example 4-32 Three-State Driver Without Registered Enable

```
// simple three-state buffer inference
#include "systemc.h"

SC_MODULE( tristate_ex5 ) {
  sc_in<bool> control;
  sc_in<sc_logic> data;
  sc_out<sc_logic> ts_out;
  sc_in_clk clk;
```

```
  sc_signal<sc_logic> temp;

  // Method for three-state driver
  void tristate_fcn () {
    if (control.read())
      ts_out.write(temp);
    else
      ts_out.write('Z');
  }

  // Method for sequential logic
  void flop () {
    temp = data.read();
  }

  // Constructor
  SC_CTOR( tristate_ex5 ) {
    SC_METHOD( tristate_fcn);
    sensitive << control << temp ;
    SC_METHOD( flop );
    sensitive_pos << clk;

  }
};
```

Figure 4-31 Three-State Driver Without Registered Enable



Understanding the Limitations of Three-State Inference

The value of Z is valid only for the sc_logic and sc_lv data types. You can use the Z value in the following ways:

- Variable assignment
- Function call argument
- Return value

You cannot use the Z value in an expression, except for comparison with Z. Be careful when using expressions that compare with the Z value. SystemC Compiler always evaluates these expressions to false, and the pre-synthesis and post-synthesis simulation results might differ. For this reason, SystemC Compiler issues a warning when it synthesizes such comparisons.

The following example shows incorrect use of the Z value in an expression:

```
OUT_VAL = ('Z' && IN_VAL);
```

The following example shows correct use of the Z value:

```
IN_VAL =  'Z';
```

Loops

SystemC Compiler supports for loops, while loops, and do-while loops for synthesis. For RTL synthesis, SystemC Compiler unrolls all loops by default. Therefore, all loops must be unrollable. Do not use the `unroll` compiler directive that is used for behavioral synthesis; it is ignored for RTL synthesis.

Loop Unrolling Criteria

To make a loop unrollable by SystemC Compiler, adhere to the following criteria for creating loops:

- A loop index must be an integer type. Valid types are char, short, int, long, sc_int, sc_bigint, and the unsigned version of these types.
- The loop index initial value must resolve to a constant at compile time.
- The loop index initial assignment cannot be in a conditional branch that may or may not be executed.
- The valid loop index operations are add, subtract, increment, and decrement.
- The valid loop condition test relational operators are <, <=, >, >=, and !=. The equality operator == is not useful for a loop condition test and is not supported for synthesis.
- The loop condition test limit must be a constant value or an expression that resolves to a constant at compile time.

- Loops cannot contain switch statements that have a continue statement.
- The loop condition cannot be null or empty.

for Loop Counter

To produce a high quality of result after synthesis, declare a unique integer variable as a for loop counter. SystemC Compiler can determine the number of bits and build only the required hardware.

Example 4-33 shows a unique loop counter variable in bold. SystemC Compiler can determine that 3 bits are required, and it builds a 3-bit incrementer for variable i.

Example 4-33 Implicit Bit Size Restriction

```
for (int i=0; i < 7; i++){
       ... //loop code
}
```

If a variable is declared outside of the for loop, SystemC Compiler cannot determine the bit size, because the intended use of the variable is not known at the time of declaration. In this situation, SystemC Compiler builds hardware for the platform-specific bit size. Example 4-34 shows code (in bold) where the loop counter variable is declared outside the loop. In such a situation, SystemC Compiler infers that a variable of 32 or 64 bits is required, depending on the platform. Therefore, it is strongly recommended that you use the coding style shown in Example 4-33 instead of the code in Example 4-34.

Example 4-34 Unknown Variable Bit Size

```
int i;
...
for (i=0; i < 7; i++){
        ... //loop code
}
```

for Loop Comma Operator

The comma (,) operator in the for loop definition in Example 4-35 is not supported for synthesis.

Example 4-35 Comma (,) Operator is Not Supported in a for Loop

```
for (i=0, j=0; i < 6; i++, j++)
```

Dead Loops

A dead loop is a loop that never executes, and SystemC Compiler issues an error if your code contains a dead loop. Example 4-36 shows a dead loop.

Example 4-36 Dead Loop

```
for (int i = 0; i > 0; i++) { ... }
```

Infinite Loops

SystemC Compiler issues an error message if your code contains an infinite loop. Example 4-37 shows various infinite loops.

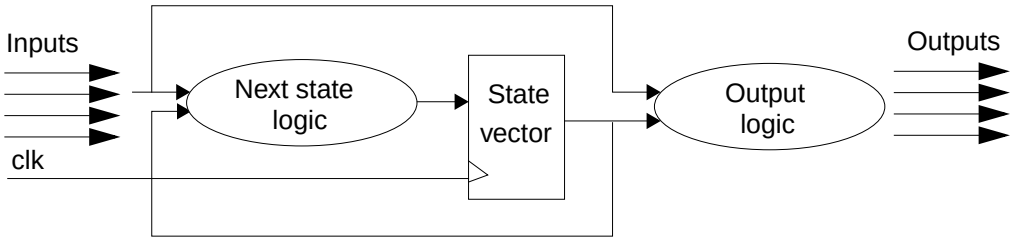
Example 4-37 Infinite Loops

```
for (int i = 1; i <= 127; i = i + 0) { ... }

for (char i = 0; i <= 127; i++) { ... }

for (char i = 0; i <= 127; i += 74) { ... }
```

State Machines

Explicitly describe state machines for RTL synthesis. Figure 4-32 shows a Mealy state machine structure.

Figure 4-32 Mealy State Machine



The diagram in Figure 4-32 has one sequential element—the state vector—and two combinational elements, the output logic and the next state logic. Although the output logic and the next state logic are separate in this diagram, you can merge them into one logic block where gates can be shared for a smaller design area.

The output logic is always a function of the current state (state vector) and optionally a function of the inputs. If inputs are included in the output logic, the state machine is a Mealy state machine. If inputs are not included, the state machine is a Moore state machine.

The next state logic is always a function of the current state (state vector) and optionally a function of the inputs.

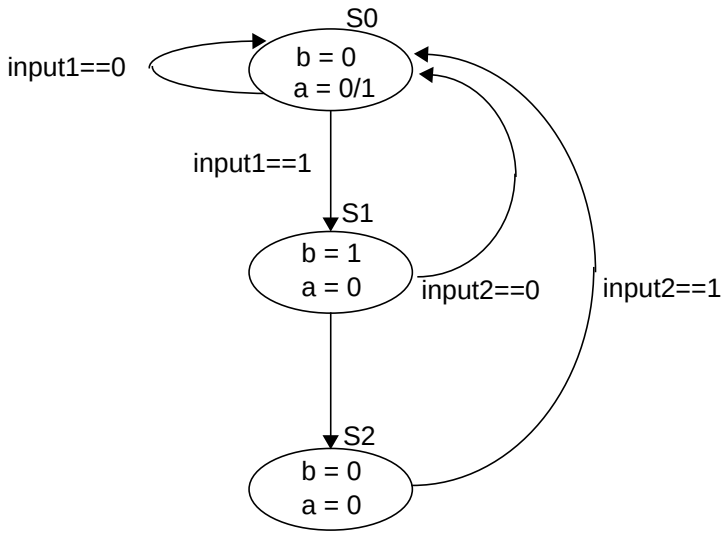
The common implementations of state machines are

- An SC_METHOD process for updating the state vector and a single common SC_METHOD process for both the output and the next state logic

- An SC_METHOD process for the state vector, an SC_METHOD process for the output logic, and a separate SC_METHOD process for the next state logic
- A Moore machine with a single process for computing and updating the next-state vector and outputs

Figure 4-33 shows a state diagram that represents an example state machine, where a and b are outputs.

Figure 4-33 Finite State Machine State Diagram



State Machine With a Common Computation Process

Example 4-38 shows the state machine (represented in Figure 4-33) with a common SC_METHOD process for computing the output and next-state logic.

Example 4-38 State Machine With Common Computation Process

```
SC_MODULE(ex_fsm_a){
  sc_in_clk clk;
  sc_in<bool> rst, input1, input2;
  sc_out<bool> a, b;

  sc_signal<state_t> state, next_state;

  void ns_op_logic();
  void update_state();

  SC_CTOR(ex_fsm_a){
    SC_METHOD(update_state);
    sensitive_pos << clk;
    SC_METHOD(ns_op_logic);
    sensitive << state << input1 << input2;
  }
};

enum state_t { // enumerate states
   S0, S1, S2
};

void ex_fsm_a::update_state()  {
  if (rst.read() == true)
      state = S0;
  else
      state = next_state;
}

void ex_fsm_a::ns_op_logic()  {
// Determine next state and output logic
  switch(state)  {
    case S0:
```

```
            b.write(0);
            if (input1.read() || input2.read())
               a.write(1);
            else
               a.write(0);
            if (input1.read() == 1)
               next_state = S1;
            else
               next_state = S0;
            break;
      case S1:
            a.write(0);
            b.write(1);
            if (input2.read() == 1)
               next_state = S2;
            else
               next_state = S0;
            break;
      case S2:
            a.write(0);
            b.write(0);
            next_state = S0;
            break;
      default:
            a.write(0);
            b.write(0);
            next_state = S0;
            break;
    }
  }
```

State Machine With Separate Computation Processes

Example 4-39 shows the state machine (represented in Figure 4-33) with separate SC_METHOD processes for computing the output and next-state logic.

Example 4-39 State Machine With Separate Processes

```
SC_MODULE(fsm_b){
  sc_in_clk clk;
  sc_in<bool> rst, input1, input2;
  sc_out<bool> a, b;

  sc_signal<state_t> state, next_state;

  void ns_op_logic();
  void output_logic();
  void update_state();

  SC_CTOR(fsm_b){
    SC_METHOD(update_state);
    sensitive_pos << clk;
    SC_METHOD(ns_logic);
    sensitive << state << input1 << input2;
    SC_METHOD(output_logic);
    sensitive << state << input1 << input2;
  }
};

enum state_t { // enumerate states
  S0, S1, S2
};

void fsm_b::update_state()  {
    if (rst.read() == true)
      state = S0;
    else
      state = next_state;
}

void fsm_b::ns_logic()  {  // Determine next state
```

```
  switch(state)  {
    case S0:
         if (input1.read() == 1)
            next_state = S1;
         else
            next_state = S0;
         break;
    case S1:
         if (input2.read() == 1)
            next_state = S2;
         else
            next_state = S0;
         break;
    case S2:
         next_state = S0;
         break;
    default:
         next_state = S0;
         break;
  }
}

void fsm_b::output_logic()  // determine outputs
{
  a.write(state == S0 && (input1.read()
          || input2.read() ) );
  b.write(state == S1);
}
```

Moore State Machine

Example 4-40 shows a Moore state machine with a single SC_METHOD process for computing and updating the output and next-state logic.

Example 4-40 Moore State Machine

```
/***ex_fsm_c.h***/
SC_MODULE(ex_fsm_c){

  sc_in_clk clk;
  sc_in<bool> rst, input1, input2;
  sc_out<bool> a, b;

  sc_signal<state_t> state;

  void update_state();

  SC_CTOR(ex_fsm_c){
    SC_METHOD(update_state);
    sensitive_pos << clk;
  }
};

/***ex_fsm_c.cpp***/
#include <iostream.h>
#include "systemc.h"
#include "fsm_types.h"
#include "ex_fsm_c.h"

void ex_fsm_c::update_state()  {
  if (rst.read() == true) {
    b.write(0);
    a.write(0);
    state = S0;
  } else {
    switch(state) {
    case S0:
      b.write(0);
      if (input1.read() || input2.read())
```

```
      a.write(1);
        else
      a.write(0);
        if (input1.read() == 1)
      state = S1;
        break;
     case S1:
        a.write(0);
        b.write(1);
        if(input2.read() == 1)
      state = S2;
        break;
     case S2:
        a.write(0);
        b.write(0);
        state = S0;
        break;
     }
   }
}
```

Defining a State Vector Variable

You can use the `state_vector` compiler directive to label a variable in your SystemC description as the state vector for a finite state machine. This allows SystemC Compiler to extract the labeled state vector from the SystemC description to use in reports and other output. For details about using this compiler directive, see “State Vector Compiler Directive” on page A-6.

A

Compiler Directives

This appendix provides a list of the compiler directives you can use for RTL synthesis with SystemC Compiler. It contains the following sections:

- Synthesis Compiler Directives
- C/C++ Compiler Directives

Synthesis Compiler Directives

To specify a compiler directive in your SystemC code, insert a comment in which the first word is either `synopsys` or `snps`. You can use either a multiple-line comment enclosed in /* and */ characters or a single-line comment beginning with two slash (//) characters.

Table A-1 lists the compiler directives in alphabetical order.

Table A-1 SystemC Compiler Compiler Directives

Compiler directive	Details
/* snps dont_infer_multibit *signal_name_list* */	page A-3
/* snps dont_infer_mux *signal_name_list* */	page A-4
/* snps enum */	page A-8
/* snps full_case */	page A-5
/* snps infer_multibit *signal_name_list* */	page A-3
/* snps infer_mux *signal_name_list* */ /* snps infer_mux */	page A-4
/* snps line_label *string* */	page A-3
/* snps parallel_case */	page A-6
/* snps state_vector string */	page A-6
/* snps synthesis_off */ and /* snps synthesis_on */	page A-9
/* snps unroll */	page A-5

Line Label Compiler Directive

Use the `line_label` compiler directive to label a loop or a line of code. In SystemC Compiler-generated reports, the label is reflected in the report hierarchy. You can also use a label with a command that sets contingencies, such as the `set_cycles` command. For example,

```
my_module2 :: entry {
         // Synopsys compiler directive
     while (true) { //synopsys line_label reset_loop2
       ...
       wait();
       ...
       wait();
     }
}
```

Instead of the `line_label` compiler directive, you can use the C/C++ line label, described in “C/C++ Line Label” on page A-9.

Multibit Inference Compiler Directives

To infer multibit for individual ports or signals, add the `infer_multibit` compiler directive to individual port or signal declarations in the SystemC description, using the following syntax:

```
sc_out<sc_int<n> > port_name; /*synopsys infer_multibit
"port_name"*/
```

Example 4-24 on page 4-38 is a code example that uses this compiler directive.

To prevent multibit inference for individual ports or signals, add the `dont_infer_multibit` compiler directive to individual port or signal declarations in the SystemC description, using the following syntax:

```
sc_out<sc_int<n> > port_name; /*synopsys
dont_infer_multibit "port_name"*/
```

Example 4-25 on page 4-40 shows a code example that uses this compiler directive.

Multiplexer Inference Compiler Directives

To infer a multiplexer for a switch…case statement, add the `infer_mux` compiler directive as the first line of code in the switch statement, using the following syntax:

```
switch (var) {
  //synopsys infer_mux
  ...
}
```

Example 4-28 on page 4-47 shows a code example that uses this compiler directive.

To infer a multiplexer for one or more switch…case statements within a block of code, add a line label to the block of code and use the `infer_mux` compiler directive, using the following syntax:

```
//synopsys infer_mux "label_1"

label_1: switch (var) {
   ...
}
```

Example 4-26 on page 4-43 shows a code example that uses this compiler directive.

Loop Unrolling Compiler Directive

Loops are unrolled by default for RTL synthesis with SystemC Compiler. Therefore, the `unroll` compiler directive used for behavioral synthesis with SystemC Compiler is not necessary, and it is ignored for RTL synthesis.

switch…case Compiler Directives

You can create multiple branching paths in logic with a switch…case statement.

Full Case

If you do not specify all possible branches of a switch statement but you know that one or more branches can never occur, you can declare a switch statement as full case with the `full_case` compiler directive. For example,

```
switch(state)  { //synopsys full_case
  case S0:
       if (input1.read())
          next_state = S1;
       else
          next_state = S0;
       break;
  case S1:
       if (input2.read())
          next_state = S2;
       else
          next_state = S0;
       break;
  case S2:
```

```
        next_state = S0;
        break;
    default:
        next_state = S0;
        break;
}
```

Example 4-17 on page 4-26 shows a code example that uses this compiler directive.

Parallel Case

SystemC Compiler automatically determines whether a switch statement is full or parallel.

All cases of a switch statement are, by definition, mutually exclusive (parallel) in C/C++. Because of this, the `parallel_case` compiler directive used by Design Compiler and other Synopsys tools is redundant. No cases overlap by design (parallel case), so SystemC Compiler synthesizes a multiplexer, because a priority encoder is not necessary.

State Vector Compiler Directive

The `state_vector` directive allows you to define the state vector of a finite state machine (and its encoding) in a SystemC description. It labels a variable in a SystemC description as the state vector for a finite state machine.

The syntax for the `state_vector` directive is

```
// synopsys state_vector vector_name
```

where *vector_name* is the variable for the state vector. This declaration allows SystemC Compiler to extract the labeled state vector from the SystemC description. Example A-1 shows one way to use the `state_vector` directive.

Note:

Do not define two `state_vector` directives in one module. Although SystemC Compiler does not issue an error message, it recognizes only the first `state_vector` directive and ignores the second.

Example A-1 Using the state_vector Compiler Directive

```
SC_MODULE( state_vector) {
  sc_in<sc_uint<2> > in1;
  sc_in_clk clock;
  sc_out<sc_uint<2> > out;

  sc_signal<sc_uint<2> > state;//snps state_vector state
  sc_signal<sc_uint<2> > next_state;

  void f1 ();
  void f2 ();
  void state_register ();

  SC_CTOR( state_vector ) {
    SC_METHOD( f1);
    sensitive (in1);
    sensitive (state);
    SC_METHOD( f2);
    sensitive (state);
    SC_METHOD(state_register);
    sensitive_pos << clock;
  }
};

void state_vector:: f1 ()
{
  switch (state.read ().to_uint ()) {
  case 0:
```

```
    next_state = (in1.read () +1) % 4;
    break;
  case 1:
    next_state = (in1.read ()+2) % 4;
    break;
  case 2:
    next_state = (in1.read ()+4) % 4;
    break;
  case 3:
    next_state = (in1.read ()+8) % 4;
    break;
  }
}

void state_vector:: state_register ()
{
  state = next_state;
}

void state_vector:: f2 ()
{
  out = state;
}
```

Enumerated Data Type Compiler Directive

The `enum` compiler directive is not used by SystemC Compiler. Use the C/C++ enum construct instead, as described in “Using Enumerated Data Types” on page 3-15.

Synthesis Off and On

The `synthesis_off` and `synthesis_on` compiler directives can be used to isolate simulation-specific code and prevent the code from being interpreted for synthesis. For example,

```
/* synopsys synthesis_off */
... //Simulation-only code
/* snps synthesis_on */
```

Use the C language #ifdef described in “C/C++ Conditional Compilation” on page A-10 instead of the `synthesis_off` and `synthesis_on` directives.

C/C++ Compiler Directives

You can use C/C++ compiler directives instead of or in addition to the equivalent `synopsys` compiler directives.

C/C++ Line Label

Use the C line label instead of the `line_label` compiler directive. For example,

```
my_module1 :: entry {
     // C-style line label
     reset_loop1: while (true) {
       ...
       wait();
       ...
       wait();
     }
}
```

C/C++ Conditional Compilation

Use the C/C++ language #if, #ifdef, #ifndef, #elif, #else, and #endif conditional compilation directives to isolate blocks of code and prevent them from being included during synthesis.

For example,

```
//C directive
#ifdef SIM
...//Simulation-only code
#endif
```

B

Examples

This appendix provides several examples that demonstrate the basic concepts of RTL synthesis with SystemC Compiler. The files for these examples are available in the SystemC Compiler installation in the $SYNOPSYS/doc/syn/ccsc directory.

This appendix contains the following examples:

- Count Zeros Combinational Version
- Count Zeros Sequential Version
- Finite Impulse Response RTL Version
- Finite Impulse Response RTL and Behavioral Integrated Version

Count Zeros Combinational Version

This circuit is a combinational specification of a design problem. The circuit is given an 8-bit value, and it determines

- That the value contains exactly one sequence of zeros
- That the number of zeros in the sequence (if any)

The circuit must complete this computation in a single clock cycle. The input to the circuit is an 8-bit value. The circuit produces two outputs, the number of zeros found and an error indication.

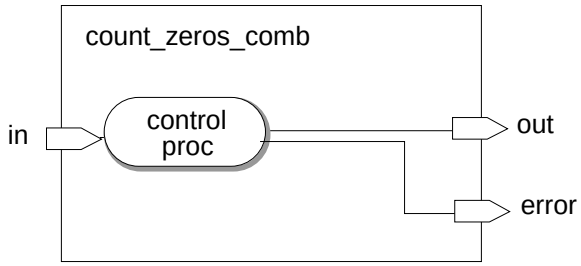
A valid value contains only one sequence of zeros. If more than one sequence of zeros is detected, the value is invalid. A value consisting of all ones is a valid value. If a value is invalid, the count of zeros is set to zero and an error is indicated. For example,

- The value 00000000 is valid, and the count is 8 zeros
- The value 1100011 is valid, and the count is 3 zeros
- The value 00111110 is invalid, and the count is zero with an error indicated

The description contains one RTL process (control_proc) and two member functions (legal and zeros). The legal function determines if the value is valid. It returns a 1-bit value of either 1, for a valid value, or 0, for an invalid value.

Figure B-1 shows the process, ports, and signals defined for this module.

Figure B-1 Count Zeros Combinational Module



The zeros function cycles through all bits of the value, counts the number of zeros, and returns the appropriate integer value. The legal and zeros functions are controlled by the control_proc process, which is sensitive to a different value on the input port.

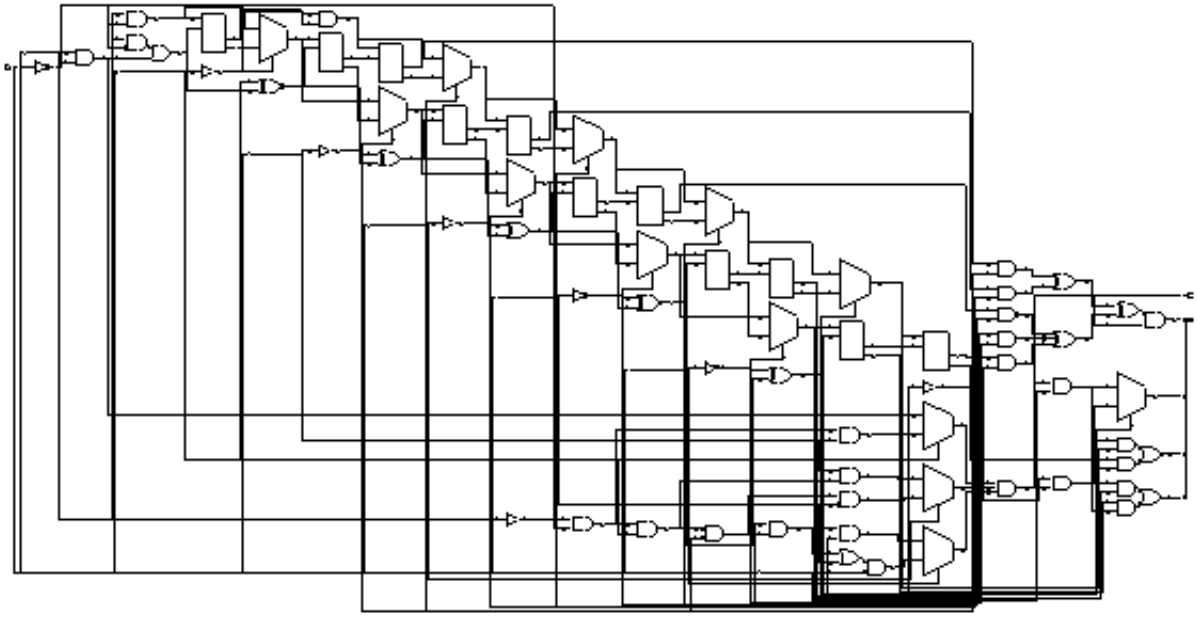
Example B-2 shows the SystemC description, and Figure B-2 shows the resulting schematic.

Example B-2 Count Zeros Combinational Version

```
/****count_zeros_comb.h file***/
#include "systemc.h"

SC_MODULE(count_zeros_comb) {
  sc_in<sc_uint<8> >  in;
  sc_out<sc_uint<4> >  out;
  sc_out<bool> error;

  bool legal(sc_uint<8> x);
  sc_uint<4> zeros(sc_uint<8> x);
  void control_proc();

  SC_CTOR(count_zeros_comb) {
    SC_METHOD(control_proc);
    sensitive << in;
  }
};


/****count_zeros_comb.cc file****/
#include "count_zeros_comb.h"

void count_zeros_comb::control_proc() {
```

```
  sc_uint<4> tmp_out;
  bool is_legal = legal(in.read());
  error.write(! is_legal);
  is_legal ? tmp_out = zeros(in.read()) : tmp_out = 0;
  out.write(tmp_out);
}

bool count_zeros_comb::legal(sc_uint<8> x) {
  bool is_legal = 1;
  bool seenZero = 0;
  bool seenTrailing = 0;
  for (int i=0; i <=7; ++i) {
    if (seenTrailing && (x[i] == 0)) {
      is_legal = 0;
      break;
    } else if (seenZero && (x[i] == 1)) {
      seenTrailing = 1;
    } else if (x[i] == 0) {
      seenZero = 1;
    }
  }
  return is_legal;
}

sc_uint<4> count_zeros_comb::zeros(sc_uint<8> x) {
  int count = 0;
  for (int i=0; i <= 7; ++i) {
    if (x[i] == 0)
      ++count;
  }
  return count;
}
```

Figure B-2 Count Zeros Combinational Schematic



Example B-3 shows the command script for synthesizing the count zeros combinational design.

Example B-3 Count Zeros Combinational Design Command Script

```
search_path  = search_path
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library      = {"*"} + target_library + synthetic_library

compile_systemc -rtl count_zeros_comb.cc -rtl_format db

compile -map_effort medium

write -hier -f db -o ./rtl_work/count_zeros_comb_gate.db

report_timing > rtl_work/count_zero_comb_timing.rpt
report_area > rtl_work/count_zero_comb_area.rpt
```

Count Zeros Sequential Version

This section provides a sequential implementation to the count zeros design described in “Count Zeros Combinational Version” on page B-2.

The circuit specification is slightly different from the specification in the combinational version. The circuit now accepts the 8-bit string serially, 1 bit per clock cycle, using the data and clk inputs. The other two inputs are

- The reset input, which resets the circuit by calling the defaults member function
- The read input, which causes the circuit to begin accepting data

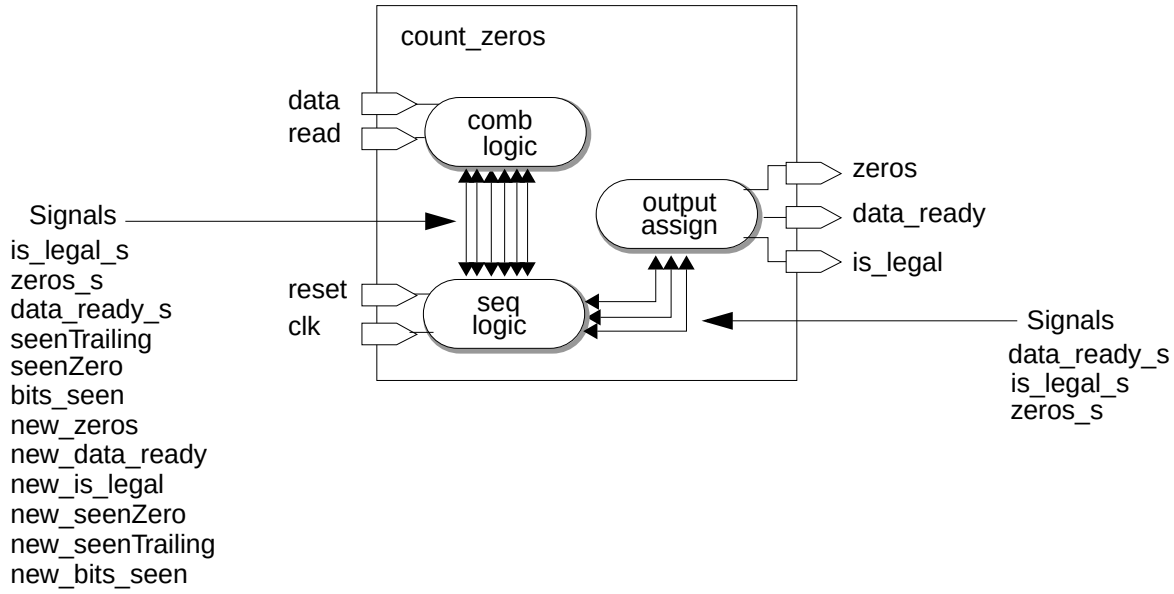
The three outputs from the circuit are

- The is_legal output, which is true if the data is a valid value
- The data_ready output, which is true when all 8 bits are processed or at the first invalid bit
- The zeros output, which is the integer value of zeros if the is_legal output is true

Example B-2 shows the SystemC description for the sequential version, and Figure B-2 shows the resulting schematic.

In this sequential version, there are three RTL processes, shown in Figure B-3. The comb_logic and output_assign processes are level-sensitive, and the seq_logic is sensitive to the positive edge of the clk input.

Figure B-3 Count Zeros Sequential Module



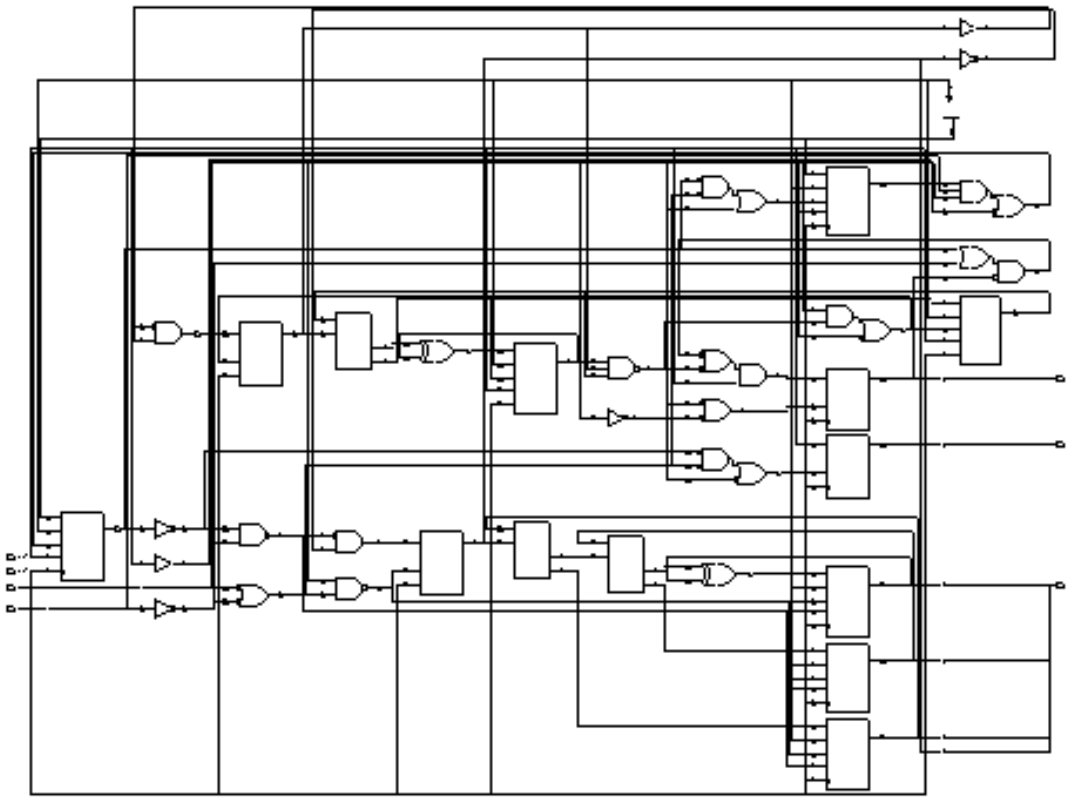
Example B-4 Count Zeros Sequential Version

```
/****count_zeros_seq.h file***/
#include "systemc.h"

#define ZEROS_WIDTH 4
#define MAX_BIT_READ 7

SC_MODULE(count_zeros_seq) {
  sc_in<bool> data, reset, read, clk;
  sc_out<bool> is_legal, data_ready;
  sc_out<sc_uint<ZEROS_WIDTH> > zeros;

  sc_signal<bool> new_data_ready, new_is_legal, new_seenZero, new_seenTrailing;
  sc_signal<bool> seenZero, seenTrailing;
  sc_signal<bool> is_legal_s, data_ready_s;
  sc_signal<sc_uint<ZEROS_WIDTH> > new_zeros, zeros_s;
  sc_signal<sc_uint<ZEROS_WIDTH - 1> > bits_seen, new_bits_seen;

  // Processes
  void comb_logic();
  void seq_logic();
  void assign_outputs();
```

```
  // Helper functions
  void set_defaults();

  SC_CTOR(count_zeros_seq) {
    SC_METHOD(comb_logic);
    sensitive << data << read << is_legal_s << data_ready_s;
    sensitive << seenTrailing << seenZero << zeros_s << bits_seen;

    SC_METHOD(seq_logic);
    sensitive_pos << clk << reset;

    SC_METHOD(assign_outputs);
    sensitive << is_legal_s << data_ready_s << zeros_s;
  }
};


/****count_zeros_seq.cc file****/#include "count_zeros_seq.h"

/*
 *  SC_METHOD: comb_logic()
 *    finds a singular run of zeros and counts them
 */
void count_zeros_seq::comb_logic() {
  set_defaults();
  if (read.read()) {
    if (seenTrailing && (data.read() == 0)) {
      new_is_legal = false;
      new_zeros = 0;
      new_data_ready = true;
    } else if (seenZero && (data.read() == 1)) {
      new_seenTrailing = true;
    } else if (data.read() == 0) {
      new_seenZero = true;
      new_zeros = zeros_s.read() + 1;
    }


    if (bits_seen.read() == MAX_BIT_READ)
      new_data_ready = true;
    else
      new_bits_seen = bits_seen.read() + 1;
  }
}

/*
 *  SC_METHOD: seq_logic()
 *    All registers have asynchronous resets
```

```
 */
void count_zeros_seq::seq_logic() {
  if (reset) {
    zeros_s = 0;
    bits_seen = 0;
    seenZero = false;
    seenTrailing = false;
    is_legal_s = true;
    data_ready_s = false;
  } else {
    zeros_s = new_zeros;
    bits_seen = new_bits_seen;
    seenZero = new_seenZero;
    seenTrailing = new_seenTrailing;
    is_legal_s = new_is_legal;
    data_ready_s = new_data_ready;
  }
}

/*
 *  SC_METHOD: assign_outputs()
 *  Zero time assignments of signals to their associated outputs
 */
void count_zeros_seq::assign_outputs() {
  zeros = zeros_s;
  is_legal = is_legal_s;
  data_ready = data_ready_s;
}

/*
 *   method: set_defaults()
 *     sets the default values of the new_* signals for the comb_logic
 *     process.
 */
void count_zeros_seq::set_defaults() {
  new_is_legal = is_legal_s;
  new_seenZero = seenZero;
  new_seenTrailing = seenTrailing;
  new_zeros = zeros_s;
  new_bits_seen = bits_seen;
  new_data_ready = data_ready_s;
}
```

Figure B-4 Count Zeros Sequential Schematic



Example B-5 shows the command script for synthesizing the count zeros sequential module.

Example B-5 Count Zeros Combinational Design Command Script

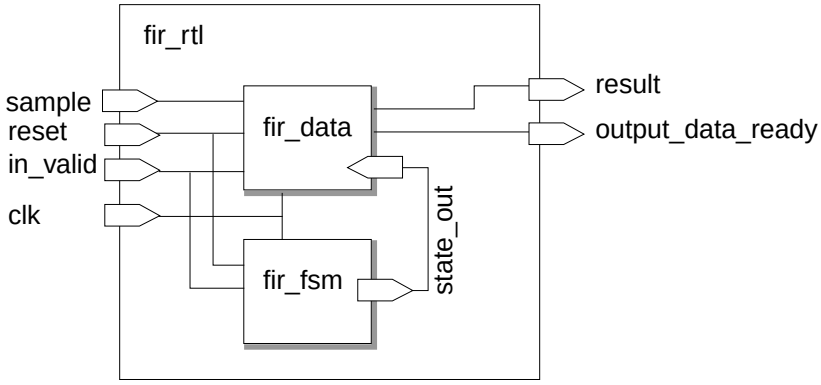
```
search_path  = search_path
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library      = {"*"} + target_library + synthetic_library

compile_systemc -rtl count_zeros_seq.cc -rtl_format db

create_clock -p 10 -name clk

compile -map_effort medium
```

```
write -hier -f db -o ./rtl_work/count_zeros_seq_gate.db

report_timing > rtl_work/count_zero_seq_timing.rpt
report_area > rtl_work/count_zero_seq_area.rpt
```

Finite Impulse Response RTL Version

The finite impulse response (FIR) filter design is a hierarchical RTL module (fir_rtl) that contains the FSM module (fir_fsm) and a data module (fir_data). Figure B-5 shows the module ports and interconnecting signal.

Figure B-5 Finite Impulse Response RTL Modules



Example B-6 shows the top-level module, Example B-7 on page B-12 shows the FSM module, and Example B-8 on page B-14 shows the data module.

Example B-6 Finite Impulse Response Top-Level RTL Module

```
/****fir_rtl.h file****/
#include <systemc.h>
#include "fir_fsm.h"
#include "fir_data.h"
```

```
SC_MODULE(fir_rtl) {

  sc_in<bool>       clk;
  sc_in<bool>       reset;
  sc_in<bool>       in_valid;
  sc_in<int>        sample;
  sc_out<bool>      output_data_ready;
  sc_out<int>       result;

  sc_signal<unsigned> state_out;

  fir_fsm  *fir_fsm1;
  fir_data *fir_data1;

  SC_CTOR(fir_rtl) {

      fir_fsm1 = new fir_fsm("FirFSM");
      fir_fsm1->clock(clk);
      fir_fsm1->reset(reset);
      fir_fsm1->in_valid(in_valid);
      fir_fsm1->state_out(state_out);

      fir_data1 = new fir_data("FirData");
      fir_data1->state_out(state_out);
      fir_data1->sample(sample);
      fir_data1->clock(clk);
      fir_data1->result(result);
      fir_data1->output_data_ready(output_data_ready);
    }
};
```

Example B-7 Finite Impulse Response FSM Module

```
/****fir_fsm.h file****/

SC_MODULE(fir_fsm) {

  sc_in<bool>      clock;
  sc_in<bool>      reset;
  sc_in<bool>      in_valid;
```

```
  sc_out<unsigned> state_out;

  // defining the states of the state machine
  enum {reset_s, first_s, second_s, third_s, output_s,
wait_s} state;

  SC_CTOR(fir_fsm)
    {
      SC_METHOD(entry);
      sensitive_pos(clock);
    };
  void entry();
};


/****fir_fsm.cpp file****/
#include <systemc.h>
#include "fir_fsm.h"

void fir_fsm::entry() {

  sc_uint<3> state_tmp;

  // reset behavior
  if(reset.read()==true) {
    state = reset_s;
  }
    // main state machine
    switch(state) {
    case reset_s:
      state = wait_s;
      state_tmp = 0;
      state_out.write(state_tmp);
      break;
    case first_s:
      state = second_s;
      state_tmp = 1;
      state_out.write(state_tmp);
      break;
    case second_s:
      state = third_s;
      state_tmp = 2;
```

```
      state_out.write(state_tmp);
      break;
    case third_s:
      state = output_s;
      state_tmp = 3;
      state_out.write(state_tmp);
      break;
    case output_s:
      state = wait_s;
      state_tmp = 4;
      state_out.write(state_tmp);
      break;

    default:
      if(in_valid.read()==true) {
      state = first_s;
      };
      state_tmp = 5;
      state_out.write(state_tmp);
      break;

    }
}
```

Example B-8 Finite Impulse Response Data Module

```
/****fir_data.h file****/

C_MODULE(fir_data) {

  sc_in<unsigned>  state_out;
  sc_in<int>       sample;
  sc_out<int>      result;
  sc_out<bool>     output_data_ready;
  sc_in<bool>      clock;

  sc_int<19> acc;
  sc_int<8> shift[16];
  sc_int<9> coefs[16];

  SC_CTOR(fir_data)
```

```
    {
      SC_METHOD(entry);
      sensitive_pos(clock);
    };
  void entry();
};


/****fir_data.cpp file****/

#include <systemc.h>
#include "fir_data.h"

void fir_data::entry()
{
#include "fir_const_rtl.h"
  sc_int<8> sample_tmp;

  sc_uint<3> state = state_out.read();

  switch (state) {
  case 0:
    sample_tmp   = 0;
    acc = 0;
    for (int i=0; i<=15; i++) {
    shift[i] = 0;}
    result.write(0);
    output_data_ready.write(false);
    break;
  case 1 :
    sample_tmp = sample.read();
    acc = sample_tmp*coefs[0];
    acc += shift[14]* coefs[15];
    acc += shift[13]*coefs[14];
    acc += shift[12]*coefs[13];
    acc += shift[11]*coefs[12];
    output_data_ready.write(false);
    break;
  case 2 :
    acc += shift[10]*coefs[11];
    acc += shift[9]*coefs[10];
    acc += shift[8]*coefs[9];
```

```
      acc += shift[7]*coefs[8];
      output_data_ready.write(false);
      break;
    case 3 :
      acc += shift[6]*coefs[7];
      acc += shift[5]*coefs[6];
      acc += shift[4]*coefs[5];
      acc += shift[3]*coefs[4];
      output_data_ready.write(false);
      break;
    case 4 :
      acc += shift[2]*coefs[3];
      acc += shift[1]*coefs[2];
      acc += shift[0]*coefs[1];
      for(int i=14; i>=0; i--) {
        shift[i+1] = shift[i];
      };
      shift[0] = sample.read();
      result.write(acc);
      output_data_ready.write(true);
      break;
    case 5 :
      // This state waits for valid input
      output_data_ready.write(false);
      break;
    default :
      output_data_ready.write(false);
      result.write(0);
    }
  }

  /****fir_const_rtl.h****/
  coefs[0] = -6;
  coefs[1] = -4;
  coefs[2] = 13;
  coefs[3] = 16;
  coefs[4] = -18;
  coefs[5] = -41;
  coefs[6] = 23;
  coefs[7] = 154;
  coefs[8] = 222;
  coefs[9] = 154;
```

```
coefs[10] = 23;
coefs[11] = -41;
coefs[12] = -18;
coefs[13] = 16;
coefs[14] = 13;
coefs[15] = -4;
```

Example B-9 shows the command script for synthesizing the hierarchical RTL finite impulse response module.

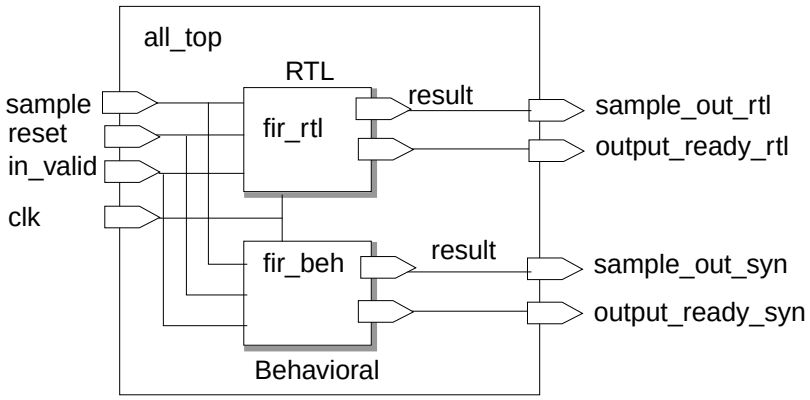
Example B-9 Finite Impulse Response RTL Hierarchical Module Command Script

```
search_path       = search_path + "$SYNOPSYS/libraries/syn"
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library      = {"*"} + target_library + synthetic_library

compile_systemc -rtl fir_fsm.cpp -rtl_format db
compile_systemc -rtl fir_data.cpp -rtl_format db
compile_systemc -rtl fir_rtl.h -rtl_format db

create_clock CLK -p 10

link

compile -map_effort medium

write -hier -f db -o fir_top_gate2.db

report_timing > rtl_work/fir_rtl_timing.rpt
report_area > rtl_work/fir_rtl_area.rpt
```

Finite Impulse Response RTL and Behavioral Integrated Version

The finite impulse response integrated RTL and behavioral design top-level module (all_top) contains both a behavioral module (fir_beh) and the hierarchical RTL module (fir_rtl) shown in “Finite Impulse Response RTL Version” on page B-11. The inputs (sample, reset, in_valid, and clk) feed into both the RTL and behavioral modules. Figure B-6 shows the inputs and the separate RTL and behavioral outputs.

Figure B-6 Finite Impulse Response RTL and Behavioral Integrated Modules



Example B-10 shows the top-level integrated module, Example B-11 shows the behavioral module, and Example B-12 shows the command script to synthesize the design.

Example B-10 Finite Impulse Response Top-Level Integrated Module

```
/****all_top.h file****/

#include <systemc.h>

#include "fir_rtl.h"
```

```
#include "fir_beh.h"

SC_MODULE(all_top) {

  sc_in<bool>       reset;
  sc_in<bool>       input_valid;

  sc_in<int>        sample;

  sc_out<int>       sample_out_rtl;
  sc_out<bool>      output_ready_rtl;
  sc_out<int>       sample_out_syn;
  sc_out<bool>      output_ready_syn;

  sc_in<bool>       clk1;

 // Instantiates RTL and behavioral models
  fir_rtl *fir_rtl1;
  fir_beh *fir_beh1;

  SC_CTOR(all_top) {

      fir_rtl1 = new fir_top("firTOP");
      fir_rtl1->reset(reset);
      fir_rtl1->in_valid(input_valid);
      fir_rtl1->sample(sample);
      fir_rtl1->result(sample_out_rtl);
      fir_rtl1->output_data_ready(output_ready_rtl);
      fir_rtl1->clk(clk1);

      fir_beh1 = new fir("FIR");
      fir_beh1->reset(reset);
      fir_beh1->input_valid(input_valid);
      fir_beh1->sample(sample);
      fir_beh1->result(sample_out_syn);
      fir_beh1->output_data_ready(output_ready_syn);
      fir_beh1->CLK(clk1);
      };
};
```

Example B-11 Finite Impulse Response Behavioral Module

```
/****fir_beh.h file****/

C_MODULE(fir_beh) {

  sc_in<bool>  reset;
  sc_in<bool>  input_valid;
  sc_in<int>   sample;
  sc_out<bool> output_data_ready;
  sc_out<int>  result;
  sc_in_clk    CLK;

  SC_CTOR(fir_beh)
     {
      SC_CTHREAD(entry, CLK.pos());
      watching(reset.delayed() == true);
     }

  void entry();
};

/****fir_beh.cpp file****/

include <systemc.h>
#include "fir_beh.h"
#include "fir_const.h"

void fir_beh::entry() {

  sc_int<8>  sample_tmp;
  sc_int<17> pro;
  sc_int<19> acc;
  sc_int<8> shift[16];


  // reset watching
  for (int i=0; i<=15; i++)
    // synopsys unroll
    shift[i] = 0;
  result.write(0);
  output_data_ready.write(false);
```

```
  wait();

  // main functionality
  fir_loop:while(1) {

    output_data_ready.write(false);
    wait_until(input_valid.delayed() == true);

    sample_tmp = sample.read();
    acc = sample_tmp*coefs[0];

    for(int i=14; i>=0; i--) {
      // synopsys unroll
      acc += shift[i]*coefs[i+1];
    };


    for(int i=14; i>=0; i--) {
      // synopsys unroll
      shift[i+1] = shift[i];
    };
    shift[0] = sample_tmp;
    // write output values
    result.write(acc);
    output_data_ready.write(true);
    wait();
  };
}


/****fir_const.h****/
static const sc_int<9> coefs[16] = {
     -6, -4, 13, 16,
     -18, -41, 23, 154,
     222, 154, 23, -41,
     -18, 16, 13, -4};
```

Example B-12 Finite Impulse Response Integrated Module Command Script

```
search_path       = search_path + "$SYNOPSYS/libraries/syn"
target_library    = {"tc6a_cbacore.db"};
synthetic_library = {"dw01.sldb","dw02.sldb"}
link_library      = {"*"} + target_library + synthetic_library

/*********************************************************/
/* Behavioral Level                                      */
/*********************************************************/

bc_enable_analysis_info    = "false"
effort_level               = medium
io_mode                    = super
top_unit                   = "fir_beh"

sh date
compile_systemc top_unit + ".cpp"

create_clock CLK -p 10

bc_time_design

schedule -io io_mode -effort effort_level

compile

write -hier -f db -o top_unit + "_gate.db"


/*********************************************************/
/* RTL Level                                             */
/*********************************************************/
sh date

compile_systemc -rtl fir_fsm.cpp -rtl_format db
compile_systemc -rtl fir_data.cpp -rtl_format db
compile_systemc -rtl fir_rtl.h -rtl_format db

create_clock CLK -p 10

link

compile -map_effort medium

write -hier -f db -o fir_rtl_gate.db
```

```
/*********************************************************/
/* Integration Level                                     */
/*********************************************************/
sh date
compile_systemc -rtl all_top.h -rtl_format db

read -f db fir_rtl_gate.db
read -f db fir__beh_gate.db

current_design all_top
link

write -hier -f db -o all_top_gate.db

report_timing > rtl_work/timing.rpt
report_area > rtl_work/area.rpt

sh date
```

Index

Symbols

#elif compiler directive A-10
#else compiler directive A-10
#endif compiler directive A-10
#if compiler directive A-10
#ifdef compiler directive A-10
#ifdef, using 3-2
#ifndef compiler directive A-10

A

abstraction level
  behavioral 1-4
  choosing for synthesis 1-4
  RTL 1-4
aggregate data type 3-15
analyze command 1-14
area report 1-15
assignment
  ports and signals 2-18
  variables 2-20
asynchronous
  low set or reset D latch 4-30
  reset 2-15
  set or reset D flip-flop 4-4
  set or reset JK flip-flop 4-11

B

behavioral
  design attributes 1-4, 1-5
  integrated with RTL 1-19, 2-36
  model 1-4
  process 2-4
  synthesis flow 1-3
behavioral and RTL
  integrated module synthesis 1-21
bit vector operators 3-12
bits
  reading and writing 2-18
buffer
  simple three-state 4-50
  three-state with register 4-52

C

C/C++
  compiler directives A-9
  data types 3-16
  line label A-9
  nonsynthesizable constructs
    C/C++ 3-5
  synthesizable subset 3-3

clock
  creation 1-14, 1-18, 1-23
  defining 2-15
  period setting 1-14, 1-18
combinational logic 2-13
comma operator, for loops 4-58
command
  entry 1-11
    dc_shell 1-11
    scripts 1-11
  order 1-10
command flow 1-10
  hierarchical RTL module 1-16
  integrated RTL and behavioral 1-19
  multiple RTL module 1-16
  single RTL module 1-10
commands
  analyze 1-14
  compile 1-14, 1-22, 1-23, 1-24
    gate-level netlist 1-19
  compile_systemc 1-11, 1-17, 1-21
    errors 1-11
    Verilog netlist 1-13
  create_clock 1-14, 1-18, 1-21, 1-23, 1-24
  current_design 1-17
  elaborate 1-14
  include 1-11
  link 1-17, 1-18
  read 1-23
  report_area 1-15
  report_timing 1-15
  write 1-13
    elaborated .db 1-13
    gate-level netlist 1-15
    HDL simulation file 1-15
compile command 1-14
compile_systemc command 1-11
  errors 1-11
  hierarchial design 1-17
  integrated RTL and behavioral design 1-21
  multiple RTL modules 1-17
  Verilog netlist 1-13
compiler directives A-2
  #elif, #else, #endif A-10
  #if, #ifdef, #ifndef A-10
  C/C++ A-9
  dont_infer_multibit 4-37
  full_case A-5
  infer_multibit 4-37, A-3
  infer_mux A-4
  line_label A-3, A-9
  parallel_case A-6
  state_vector 4-66, A-6
  synthesis_off A-9
  synthesis_on A-9
  unroll A-5
  using #ifdef 3-2
  using synthesis_off 3-2
compiling gate-level netlist 1-14, 1-19, 1-22
constructor
  module 2-16
constructs
  nonsynthesizable C/C++ 3-5
  nonsynthesizable SystemC 3-4
control logic 1-6
count zeros combinational example 2-22, B-2
count zeros sequential example 2-24, B-6
create_clock command 1-14, 1-18, 1-23
creating
  HDL netlist 1-13
  hierarchical module 2-28
  hierarchical RTL module 2-30
  integrated RTL and behavioral 2-36
  module 2-5
  multiple modules 2-28
  process 2-12
  single module 2-22
current_design command 1-17

D

D flip-flop
  simple 4-2
  with asynchronous set or reset 4-4
  with synchronous set or reset 4-8
D latch
  inferring 4-16
  with low asynchronous set or reset 4-30
data member
  of a module 3-17
  variables 2-11
data path 1-6
data refinement 3-8
  C/C++ 3-3
  recommendation 3-20
  SystemC 3-3
data types
  aggregate 3-15
  C++ 3-16
  enumerated 3-15
  nonsynthesizable 3-9
  ports 2-8
  recommended for synthesis 3-9, 3-10
  signal 2-10
  synthesizable 3-8
  SystemC operators 3-12
database
  behavioral 1-22
  creating .db file 1-12
  GTECH 1-12
  hierarchical design 1-17
  multiple RTL module design 1-17
  RTL 1-21
  RTL and behavioral 1-23
dc_shell
  command entry 1-11
  starting 1-11
dead loop 4-58
define
  control logic 1-6
  data path 1-6
  FSM 1-6
  I/O 1-6
  level-sensitive 2-13
  module 2-2, 2-5
  process 2-12
  sensitivity list 2-13
design
  behavioral attributes 1-4, 1-5
  command order 1-10
  RTL attributes 1-4
DesignWare library 1-9
directories
  rtl_work 1-13
dont_infer_multibit compiler directive 4-37

E

edge-sensitive
  clock 2-15
  inputs 2-14
  process 2-14
  sensitivity list 2-14
elaborate command 1-14
entering commands 1-11
enumerated data type 3-15
examples
  count zeros combinational 2-22, B-2
  count zeros sequential 2-24, B-6
  FIR integrated RTL and behavioral 2-36, B-18
  FIR RTL 2-30, B-11
excluding
  nonsynthesizable code 3-2
  simulation-only code 3-3

F

files
  database .db file 1-12
  gate-level netlist 1-9
  RTL HDL 1-9

FIR integrated RTL and behavioral example 2-36, B-18
FIR RTL example 2-30, B-11
flip-flop
  inference 4-2
  inferring D with asynchronous set or reset 4-4
  inferring D with synchronous set or reset 4-8
  inferring JK 4-10
  inferring JK with asynchronous set or reset 4-11
  inferring JK with synchronous set or reset 4-10
  simple D 4-2
for loop
  comma operator 4-58
format
  GTECH 1-12
FSM
  definition 1-6
  description 4-59
full_case compiler directive A-5
functions
  member 2-16

G

gate-level
  writing netlist 1-15, 1-19
  writing simulation file 1-15
GTECH database 1-12

H

HDL
  creating netlist 1-13
  RTL file 1-9
hdlin_enable_presto variable 1-12, 1-14
hdlin_infer_multibit variable 4-38
hdlin_unsigned_integers variable 1-12, 1-14
header file
  module 2-5
hierarchical design 1-17
  command flow 1-16
hierarchical module
  creating 2-28
  RTL 2-30

I

implementation
  module 2-17
include command 1-11
incomplete sensitivity list 2-14
infer_multibit compiler directive 4-37, A-3
infer_mux compiler directive A-4
inference
  flip-flop 4-2
  latch 4-16
  multibit 4-37
  multiplexer 4-42
  register 4-2
  three-state 4-50
inferring
  D latch 4-16
  JK flip-flop 4-10
inout port 2-7
input port 2-7
inputs
  edge-sensitive 2-14
  level-sensitive 2-13
inputs to SystemC Compiler 1-7
integrated RTL and behavioral 1-21
  command flow 1-19
  creating 2-36
  synthesis 1-21

J

JK flip-flop
  inferring 4-10
  with asynchronous set or reset 4-11
  with synchronous set or reset 4-10

L

label
  C/C++ line label A-9
  line_label compiler directive A-3
latch
  avoiding 4-16
  D with low asynchronous set or reset 4-30
  inference 4-16
  SR 4-17
level-sensitive 2-13
  inputs 2-13
  sensitivity list 2-13
library
  DesignWare 1-9
  synthetic 1-7, 1-9
  technology 1-7, 1-8
limitations
  multiplexer inference 4-49
  register inference 4-33
  sensitivity lists 2-16
  three-state inference 4-55
line_label compiler directive A-3, A-9
link command 1-17, 1-18
logic
  combinational 2-13
  sequential 2-14
loops 4-56
  dead 4-58
  unrolling A-5

M

member functions 2-16
module
  constructor 2-16
  contents 2-2
  creating 2-5
  creating multiple 2-28
  creating single 2-22
  data members 3-17
  defining 2-2
  header file 2-5
  implementation 2-17
  ports 2-7
  SC_MODULE 2-7
  signal communication 2-9
  syntax 2-6
multibit
  inference 4-37
multiple modules
  creating 2-28
multiple RTL modules 1-17
multiplexer inference 4-42
  limitations 4-49

N

netlist
  gate_level 1-9
new features in this release 1-xxii
nonsynthesizable
  data types 3-9
nonsynthesizable code
  excluding 3-2

O

operators
  SystemC bit vector 3-12
  SystemC data type 3-12
output port 2-7
outputs from SystemC Compiler 1-7

P

parallel_case compiler directive A-6
period
  clock 1-14, 1-18
port type
  sc_in 2-7
  sc_inout 2-7
  sc_out 2-7
ports 2-7
  assignment 2-18
  data types 2-8
  read 2-17
  read bits 2-18
  syntax 2-8
pragma
  See compiler directive
process
  behavioral 2-4
  creating in a module 2-12
  edge-sensitive 2-14
  execution 2-3
  level-sensitive 2-13
  read and write 2-4
  registering 2-3
  RTL 2-4
  SC_CTHREAD 2-4
  SC_METHOD 2-4
  SC_THREAD 2-4
  sensitivity list 2-4, 2-13
  trigger 2-4
  types 2-4

R

read
  from ports 2-4
  port bits 2-18
  ports 2-17
  signal bits 2-18
  signals 2-17
read command 1-23
recommendation
  data refining 3-20
refine
  chip-level block 1-6
  data 3-8
  data recommendation 3-20
  overview 1-6
  pure C/C++ 1-6
register
  inference 4-2
  inference limitations 4-33
registering a process 2-3
report_area command 1-15
report_timing command 1-15
reports
  area 1-15
  timing 1-15
reset
  asynchronous 2-15
rolled and unrolled loops A-5
RTL
  design attributes 1-4
  design description 1-8
  hierarchical module 1-16, 2-30
  integrated module 1-19
  integrated with behavioral 1-19
  model 1-4
  process 2-4
  synthesis command flow 1-10
  synthesis commands for integrated modules 1-19
  synthesis commands for multiple modules 1-16
  synthesis flow 1-3
RTL and behavioral
  integrated module synthesis 1-21
RTL integrated with behavioral 2-36
rtl_work directory 1-13

S

saving
  database .db file 1-13
SC_CTHREAD process 2-4
sc_in port type 2-7
sc_inout port type 2-7
SC_METHOD process 2-4
SC_MODULE 2-7
  syntax 2-6
sc_out port type 2-7
SC_THREAD process 2-4
sensitive() 2-13
sensitive_neg() 2-13
sensitive_pos() 2-13
sensitivity list 2-13
  defining 2-13
  edge-sensitive 2-14
  incomplete 2-14
  level-sensitive 2-13
  limitations 2-16
sequential logic 2-14
signal 2-9
  assignment 2-18
  communication between processes 2-4
  data types 2-10
  read 2-17
  read and write 2-4
  read bits 2-18
  syntax 2-10
simulation
  gate-level HDL file 1-15
simulation-only code
  excluding 3-3
snps compiler directive A-2
source code
  RTL 1-8
SR latch 4-17
start
  dc_shell 1-11
state machine
  description 4-59
state_vector compiler directive 4-66, A-6
subset
  synthesizable 3-2
synchronous
  set or reset D flip-flop 4-8
  set or reset JK flip-flop 4-10
synopsys compiler directive A-2
syntax
  module 2-6
  port 2-8
  signal 2-10
synthesis
  choosing abstraction level 1-4
  creating .db file 1-12
  creating .db for behavioral module 1-22
  creating .db for hierarchical design 1-17
  creating .db for integrated RTL and
      behavioral module 1-23
  creating .db for multiple RTL modules 1-17
  creating .db for RTL module 1-21
  creating HDL netlist 1-13
  define control logic 1-6
  define data path 1-6
  define FSM 1-6
  define I/O 1-6
  excluding nonsynthesizable subset 3-2
  recommended data types 3-9, 3-10
  refining functional model 1-6
  refining requirements 1-6
  synthesizable subset 3-2
synthesis flow 1-3
  behavioral 1-3
  commands 1-10
  commands for integrated modules 1-19
  commands for multiple modules 1-16
  RTL 1-3
synthesis_off compiler directive A-9
synthesis_off, using 3-2
synthesis_on compiler directive A-9

synthesizable
  data types 3-8
  subset 3-5
synthetic library 1-7, 1-9
  location 1-9
synthetic_library variable 1-9
SystemC
  data type operators 3-12
    bitvector 3-12
  nonsynthesizable constructs 3-4
  RTL description 1-8
  synthesizable subset 3-3

T

target_library variable 1-8
technology library 1-7, 1-8
  location 1-8
three-state
  inference 4-50
  inference limitations 4-55
  simple buffer 4-50
  with register 4-52
timing report 1-15
triggering a process 2-4

U

unroll compiler directive A-5
using scripts 1-11

V

variables
  assignment 2-20
  data members 2-11
  hdlin_enable_presto 1-12, 1-14
  hdlin_infer_multibit 4-38
  hdlin_unsigned_integers 1-12, 1-14
  read and write 2-4
  synthetic_library 1-9
  target_library 1-8

W

what’s new in this release 1-xxii
write
  elaborated .db 1-13
  gate-level netlist 1-15, 1-19
  gate-level netlist for synthesis 1-14
  HDL simulation file 1-15
  to ports 2-4